Graph-Based Whole Genome Phylogenomics

Masaki Stanley Fujimoto

Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Physical Sciences and Mathematics Commons

BYU ScholarsArchive Citation

Fujimoto, Masaki Stanley, "Graph-Based Whole Genome Phylogenomics" (2020). Theses and Dissertations. 8461.
https://scholarsarchive.byu.edu/etd/8461

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Graph-Based Whole Genome Phylogenomics

Masaki Stanley Fujimoto

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Mark J. Clement, Chair
Quinn O. Snell
Seth Bybee
Mike Jones

Department of Computer Science
Brigham Young University

Copyright © 2020 Masaki Stanley Fujimoto
All Rights Reserved
ABSTRACT

Graph-Based Whole Genome Phylogenomics

Masaki Stanley Fujimoto
Department of Computer Science, BYU
Doctor of Philosophy

Understanding others is a deeply human urge basic in our existential quest. It requires knowing where someone has come from and where they sit amongst peers. Phylogenetic analysis and genome wide association studies seek to tell us where we’ve come from through evolutionary history and who we are in our genetic makeup. Current methods do not address the computational complexity caused by new forms of genomic data, namely long-read DNA sequencing and an increased number of assembled genomes, that are becoming evermore abundant. To address this, we explore specialized data structures for storing and comparing genomic information. This work resulted in the creation of novel data structures for storing multiple genomes that can be used for identifying structural variations and other types of polymorphisms. Using these methods we illuminate the genetic history of organisms in our efforts to understand the world around us.

Keywords: data structures, de Bruijn Graph, genomics, next-gen sequencing, NGS read mapping, parallel programming, phylogenetics, phylogenomics, whole genome alignment, whole genome analysis
ACKNOWLEDGMENTS

My greatest thanks to my spouse, Madison Fujimoto. Thank you for your patience, encouragement and unconditional love. I couldn’t have done it without you.

I also acknowledge and give thanks the eternally optimistic support of my advisor, Dr. Mark J. Clement, who provided many opportunities, valuable feedback and was always welcoming of us in his home.

I thank my parents, Andy and Yoshie Fujimoto, for providing the love and means to discover and develop my own interests. I also thank my siblings and their families for their love and encouragement. All of which, I have learned, is rare in this world.

I have made some of my best friends while performing this work and would like to thank Dr. Paul M. Bodily, Cole A. Lyman, Dr. Anton Suvorov and Nozomu Okuda for their love, mentoring and collaborative spirits. Thanks for helping me through this.

I thank current and former members of my dissertation committee for their time and feedback: Dr. Christophe Giraud-Carrier, Dr. Quinn Snell, Dr. Seth Bybee, Dr. Mike Jones and Dr. Kent Seamons.

I would also like to thank the many professors, colleagues and friends I have made during this process: Dr. Ryan Farrell, Dr. Perry Ridge, Nick Jensen, Jonathan Dayton, Kristi Bresciano, Michael Watkins, Spring Cullen, Andrew Jacobsen, Brian Brown, Kyle Johnsen, Kaleb Olson, Brandon Pickett, Dr. Brandon Miller, Connor Anderson, Shayla Draper, Hunter Wright, Nik and Bronwyn Putnam, Drew Holman, Katie Holloman, Kevin and Lisa Keovongsa, Jon and Becca Ogden, Gary Crofts, Greta Baliff, Scott and Sarah Campbell, Phillip Stevenson, Wilson Fearn, Kim McIntire, Mike Brodie, Pei Guo, Dr. Xie Summer Xue, Camilla Sharkey, Dr. Christopher Tensemyer, Dr. Aaron Dennis, Dr. Derrall Heath and Dr. Joe Price.

Lastly, I greatly appreciate funding that has been graciously provided for this research from the BYU Computer Science Department, BYU Graduate Studies through their Graduate Research
Fellowship and from the Utah NASA Space Grant Consortium and EPSCoR through their Space Grant Fellowship.
Table of Contents

List of Figures x

List of Tables xvii

List of Listings xx

1 The Difficulties of Comparative Genomics and Phylogenetic Inference 1
   1.1 Introduction ................................................................. 1
   1.2 Outline of Work .............................................................. 1

2 GNUMAP 4.0: Space and Time Efficient NGS Read Mapping Using the FM-Index 4
   2.1 Abstract ................................................................. 4
   2.2 Background ................................................................. 4
   2.3 Methods ................................................................. 6
      2.3.1 FM-Index Implementation ......................................... 6
      2.3.2 Additional Optimizations and Bug Fixes .................... 7
      2.3.3 Datasets ............................................................ 7
      2.3.4 Benchmarking ...................................................... 9
   2.4 Results and Discussion .................................................... 10
      2.4.1 Memory Usage ....................................................... 10
      2.4.2 Run-Time ............................................................ 13
      2.4.3 Index Size on Disk ............................................... 14
      2.4.4 Parameter Tuning ............................................... 15
# The OGCleaner: Detecting False-Positive Sequence Homology

## 3.1 Abstract

## 3.2 Background

## 3.3 Methods

### 3.3.1 Construction of ground-truth training sets

### 3.3.2 Attribute selection

## 3.4 Real Data Set Construction

### 3.4.1 Library preparation and RNA-seq

### 3.4.2 Read trimming and de novo transcriptome assembly

### 3.4.3 Downstream transcriptome processing

### 3.4.4 Construction of Drosophila data set

### 3.4.5 Gene homology inference

## 3.5 Implementation

### 3.5.1 Training Data Set

### 3.5.2 Validation Data Set

### 3.5.3 Testing Data Set

### 3.5.4 Real Data Set

### 3.5.5 Miscellaneous Parameters

## 3.6 Results and Discussion

## 3.7 Conclusion

---

# Whole Genome Phylogenetic Tree Reconstruction Using Colored de Bruijn Graphs

## 4.1 Abstract

## 4.2 Introduction

## 4.3 Methods

### 4.3.1 Definitions

---

vi
List of Figures

1.1 The relative efficiency and resolution of different comparative genomics techniques with our proposed method, the Polygraph, emphasized. ........................................... 3

2.1 The peak memory usage (in gigabytes) of GNUMAP using either the hash or FM-index data structure with a particular kmer size $k$ during run-time. As can be seen, the RAM usage with the hash version increases as $k$ increases while the FM-index implementation remains the same across all sizes $k$. There are no hash results for $k > 15$ due to limitations of the hashing algorithm that does not allow for hashes to be built with $k > 15$. ................................................................. 8

2.2 The kmer count profiles of the human reference genome hg19 using $k = 11$ and $k = 15$. Differences in these plots show why the 11mer mapping process takes much longer than using 15mers. The maximum kmer occurrence threshold of 1,000 is marked on the plot, kmers with frequencies above this threshold were not used for mapping done in Table 2.1. Area under the curve shows how much of the index is based on rare or frequent kmers. The 11mer index is skewed to the right but has much more of its area concentrated near the occurrence threshold than the 15mer plot. This results in an index that is composed of many more frequent kmers that can trigger false-positive alignments. The 15mer plot is skewed much more to the right and has little area concentrated near the occurrence threshold. An index with this profile is composed of kmers that occur less often throughout the genome causing less alignments to false-positive locations. ....................................... 13
3.1 Cluster generation workflow. This is the process of generating homology and non-homology clusters for training the machine learning algorithms. 

3.2 T-SNE manifold learning dimensionality reduction applied to the true-positive OrthoDB clusters (blue, H) and generated false-positive clusters (green, NH). Instances from both classes overlap and are not easily separable suggesting that the generated clusters are similar to the true-positive clusters.

3.3 A diagram of the overall workflow of the OGCleaner. This figure shows the different steps that were used in developing our machine learning model. Arthropod phylogeny was generated in previous studies and deposited in OrthoDB. These sequences were then gathered from OrthoDB and used as our orthology and paralogy clusters. They were combined with generated non-homology clusters. The combination represents our training data set used to train the machine learning algorithms. The experimental data were assembled with proteins inferred from the assemblies. InParanoid was then used to identify putative homologs. Once putative homologs were identified they were input into the trained machine learning algorithms for classification and subsequent cluster trimming.

3.4 Training data set accuracy.

3.5 Testing data set accuracy.

3.6 Area under the Receiver Operating Characteristic curve (AUROC) for the neural network model. This shows the trade-off between sensitivity (true-positive rate) and specificity (false-positive rate) by varying classification confidence thresholds. An area under the curve (AUC) of nearly 1 shows that there is little trade-off between sensitivity and specificity.
3.7 Examples of a high quality homology (a) and false-positive homology (b) All sequences within the homology cluster (a) belong to one protein family (FAM81A1-like protein). The sequence in the false-positive homology cluster indicated by the arrow represents Aprataxin and PNK-like factor whereas other sequences represent tyrosyl-DNA phosphodiesterase.

3.8 Training data set per attribute performance using MLP

3.9 Testing data set per attribute performance using MLP

4.1 A. An example of a bubble in a Colored de Bruijn Graph with 3 colors (i.e. 3 taxa), and where $k = 3$. The colors of the vertices represent the following: gray- all colors contain the vertex, purple- Color 2 and Color 3 contain the vertex, yellow- Color 1 contains the vertex, red- Color 2 contains the vertex, and blue- Color 3 contains the vertex. In this example ACT is the startVertex and GTG is the endVertex which are both contained in all of the colors. B. The extended paths of each color between the startVertex and endVertex.

4.2 A. The Multiple Sequence Alignment (MSA) of the sequences from the bubble presented in Figure 4.1. B. The MSA’s from each bubble are concatenated into a supermatrix, from which a phylogenetic tree is constructed. C. The resulting tree from the supermatrix inferred by Maximum Likelihood.

4.3 The phylogenetic tree of 12 Drosophila species constructed using kleuren. This tree resulted from using a kmer size of 17 and required all species to contain a vertex in order for the algorithm to search for a bubble starting at that vertex; and this tree is consistent with the established tree for these 12 species.

5.1 The construction of a standard de Bruijn Graph. A) The original sequence. B) sequence broken into kmers ($k=4$) showing kmer overlap. C) A de Bruijn graph with edges formed from overlapping kmers.
5.2 Construction method for our relaxed de Bruijn Graph for two reference genome sequences. A and D are two different sequences. B and E represent the sequence broken into kmers and the graph node IDs assigned to each kmer. C is the initial relaxed de Bruijn graph containing only A. Blue nodes are unique kmers and red nodes are non-unique kmers occurring in sequence A. F is the resulting relaxed de Bruijn graph once kmers from sequence D are added. Green nodes and edges are new nodes or edges that were added to the graph. See Algorithm 5 for construction of C and Algorithm 6 for F.

5.3 Graph structure formed from a synthetic genome and synthetically generated reads after node simplification to unitigs. Graph was constructed using $k = 31$. Blue nodes are sequence from the reference genome and green are sequences from synthetically generated NGS reads. Node 7191 contains a point mutation, node 7187 contains an insertion, node 7188 contains an inversion and the structure that forms from nodes 7185, 7189, and 7190 represent a translocation.

5.4 Relaxed de Bruijn graph for two *E. coli* genomes (MG1655 and W3110) using $k = 1001$. Many graph structures appear beyond simple bubbles. Additional methods are needed for characterization of complex graph structures that form.

6.1 The Polygraph built for three input genomes. Each vertex contains sequence positions (start:end) ordered from genome 0 (top) to 2 (bottom). A $-1$ entry means a genome is not present in a vertex. Colored vertices indicate where sequence is made entirely of shared-unique k-mers with different colors indicating which genomes are present.

6.2 (a) The Polygraph identifies an inverted translocation from chromosome VIII highlighted in red and an inversion in magenta that (b) progressiveMauve does not identify. Visualized using Muave Viewer.
6.3 A portion of the Polygraph for yeast chromosome VI. Vertices store genomic coordinates as well as orientation of sequences. Edges of the graph contain a list of all genomes that traverse that edge to facilitate graph traversal algorithms. Coordinates with \(-1\) indicate a genome is not present. 

6.4 Mapping of the three genes from Ref:VI (YFL059W, YFL060C and YFL061W) on the top row and Ref:XIV (YNL333W, YNL334C and YNL335W) on the bottom in IGV [114].

6.5 Multiple genome alignment of five \textit{E. coli} genomes with three different homologous regions highlighted in red, magenta and cyan in the Mauve Viewer.

7.1 K-mer serialization packs each base into 2-bits. The left-most base in a k-mer is stored in the right-most position when serialized. Sorting serialized k-mers is done using C’s \texttt{memcmp} function resulting in a non-lexicographic sorting in nucleotide base-space.

7.2 Finding the child vertex that represents a given 4-mer by calculating the hamming weight of the child vertex presence array. The child vertex for k-mer AAGA is found by serializing the k-mer 00100000. Next, the serialized value is cast to an int (32 in this case) and is used as an index into the population array. The count of 1’ss that precedes this index (cells highlighted in orange) correspond to the index into the child vertices array for this k-mer. This count is determined by left-shifting the child vertex presence array (a) by \(256 - 32 = 224\) bits to produce (b). The hamming weight in the shifted array is found by counting the 1s left in the array. In this case 9 1s correspond to the correct index in the child vertex array (c).

7.3 Fast k-mer sorting. Given \(t = 4\) threads, the number of bits to shift is \(s = 6\). The appropriate consumer thread is determined quickly by shifting the first 4-mer prefix of the k-mer \(s\) bits and casting it as an unsigned integer \(i\) which corresponds to the appropriate consumer thread.
7.4 After insertion, all consumer threads are merged together. (a) shows child vertex presence arrays for each thread highlighting the 4-mer prefixes that each thread handles. Consumer thread joining concatenates these arrays together using logical or and storing the result in the producer thread. (b) Child vertex arrays are concatenated together as well. Different colors indicate which thread a k-mer prefix was originally assigned to. ................................................................. 90

8.1 An inversion within three synthetic genomes visualized using Mauve [23] and aligned from the PolyGraph. Note the magenta region in genome 3 that appears in the bottom half of the area designated for genome 3 in the visualizer signifying it is on the negative strand while the magenta regions in genomes 1 and 2 are on the top halves indicating the positive strand. ................................................................. 98

8.2 A translocation in three synthetic genomes visualized using Mauve [23] and aligned with the PolyGraph. See the region A denoted in all three genomes that occurs earlier in genome 1 compared to the others. ................................................................. 99

8.3 Visualization of Prasinovirus greedily extracted evolutionary splits from the PolyGraph using SplitsTree4 [66], k=17 ................................................................. 102

8.4 Trees for Prasinoviruses ................................................................. 103

8.5 Trees for the E. coli dataset colored by recognized E. coli phylogenetic groups . . 105

8.6 Visualization of E. coli greedily extracted phylogenetic network using evolutionary splits from the PolyGraph using SplitsTree4 [66], k=31 ................................................................. 105

8.7 Trees for Drosophila ................................................................. 107

8.8 Top 50 evolutionary splits by weight for Drosophila generated from SANS and the PolyGraph labeled by agreement with the canonical tree. a Looking at all evolutionary splits and their concordance with the canonical tree can result in overstating the correctness of the evolutionary splits as many of the splits only contain 1 taxa. b Only evolutionary splits with 2 or more taxa are shown showing far fewer splits that agree with the canonical tree. ................................................................. 108
8.9 Visualization of *Drosophila* greedily weakly extracted splits from the PolyGraph
generated evolutionary splits using SplitsTree4 [66], k=31  

8.10 Regions of *D. grimshawi*, *D. mojavensis* and *D. virilis* that anchor together in the
PolyGraph. An evolutionary split containing *D. grimshawi* and *D. mojavensis* and
another containing *D. grimshawi* and *D. virilis* were the highest weighted discordant
evolutionary splits (see Table 8.4).
List of Tables

2.1 GNUMAP using hashmap and FM-index comparison using the same mapping parameters. Run-time and number of mapped reads using the hashmap and FM-index versions of GNUMAP on the ART generated synthetic dataset and on the real dataset. As kmer size increases, from 7 to 10, run-time is about the same. For kmer sizes 11 and 12 run-time is much larger (see bolded rows). The run-time increase for these kmers is caused by the composition of hg19 where lots of 11 and 12mers (13mers in real dataset where 11 and 12mers didn’t finish) happen to have frequencies near the chosen threshold (1,000) resulting in more NW alignments that are costly in terms of time. For 13mers and larger kmers, the run-time decreases and eventually settles at a minimum run-time.

2.2 GNUMAP parameter tuning using the FM-index. The FM-index allows for much more flexibility in regards to parameter tuning for read mapping. Here, we compare the GNUMAP with the hashmap to the FM-index with various parameters tuned. Tuning the parameters in GNUMAP-FM greatly increases the number of reads mapped while decreasing the run-time and amount of memory and maintaining mapping accuracy.

2.3 GNUMAP hashmap size on disk for various kmer sizes and maximum kmer occurrence 1,000. Hashes with $k > 15$ were unable to be generated.

3.1 All Features that were used in order to train the machine learning algorithm. Each of these features was calculated for each of the clusters Machine learning.
3.2 The machine learning parameters used for each of the different algorithms in initial training and testing with WEKA. ....... 29

3.3 Sequence Read Archive statistics and IDs. ....... 30

3.4 Drosophila data sets. ....... 31

3.5 Confusion matrix of test instances (1835 total instances) for the neural network model. 34

3.6 Per-class performance of the test-set measured with precision, recall and F1-score with support (number of instances) for each class. ....... 34

3.7 Summary of InParanoid and HaMStR cluster filtering. The number of clusters that were kept and removed for the OD_S clusters from InParanoid and HaMStR. ... 41

3.8 Agreement with Reference Gene Phylogenies. Higher values are better for 'pos. predictive value rate' and 'true-positive rate'. The '-' character represents when OrthoMCL + The OGCleaner provided the best performance under the 'Best Method' columns. Improvements are marked in green and weaker results in red. OrthoMCL was used to generate a base set of orthology clusters. The OGCleaner was then applied to the clusters using the provided pre-trained model filtering out low-quality clusters. ....... 46

3.9 Generalized Species Tree Discordance Benchmark. Lower values are better for 'avg RF distance(genetree, speciestree)' and 'avg fraction incorrect trees'. ...... 46

3.10 Species Tree Discordance Benchmark. Higher value is better for 'avg Schlicker' and '# ortholog relations' is better. ....... 47

7.1 Time and memory usage for indexing the human genome. The BFT requires k-mer generation in a pre-process step, we use jellyfish and include those results. Both the build time and overall running time are provided for the Python set due to the large disparity between them. ....... 91

7.2 Time and memory usage for 20M queries against the human reference genome. 10M k-mers that exist and 10M that do not exist in the index. ....... 91
8.1 13 genomes from the Prasinovirus dataset ..................................................... 102
8.2 27 genomes from the *E. coli* dataset. ............................................................ 104
8.3 12 genomes used in the *Drosophila* dataset ................................................. 107
8.4 The top 30 evolutionary splits by weight calculated from the PolyGraph using the modified SANS algorithm. Most of the top weights evolutionary splits contain a single species. ................................................................. 111
List of Listings

7.1 Merging function example in Python that stores the position of each k-mer occurrence in a list. prev_value and new_value are both lists that initially hold a single genomic position. As the same k-mer is inserted, the positions accumulate. 91

7.2 Merging function example in C++ for k-mer counting. The map is instantiated to store ints as values. When k-mers are inserted they are all inserted with a value of 1. As the same k-mer is inserted, the values are summed together resulting in k-mer counting. 91
Chapter 1

The Difficulties of Comparative Genomics and Phylogenetic Inference

1.1 Introduction

Comparing sequences of DNA is a core bioinformatic task that is still quite difficult. It is important in measuring gene activation levels, identifying mutations and understanding relatedness of organisms. Sequence comparison is difficult and requires specialized algorithms and data structures. The computational complexity of DNA sequence comparison has shaped the types of analyses that are commonly used.

In this work, we explore these limitations and provide improvements that take advantage of improving DNA sequencing technologies, namely long-read sequencing and the increasing availability of whole genome assemblies. We make use of specialized and novel data structures designed for storing DNA and sequence comparison. We then use these data structures to conduct phylogenetic inference and genome wide association studies (GWAS).

1.2 Outline of Work

Chapter 2 begins with a modernization of the GNUMAP NGS read mapping algorithm by integrating the FM-index as the genome indexing algorithm used during the mapping process [41]. This work allowed us to explore mapping long reads to a reference genome to help identify haplotypes that exist amongst a population. While memory usage reduced significantly using the new data structure, we found that reference bias resulted in low mapping results and sought additional methods.

In Chapter 3, we turn our focus towards phylogenetic tree inference where we infer evolutionary relatedness of organisms through gene sequence comparison. Here, we examine gene clustering
algorithms that produce groupings of genes that have enough sequence similarity to infer their relatedness. These clusters are formed in a pair-wise comparison of gene sequences which can lead to a cluster with desirable qualities at a pair-wise sequence level but with pair sequence alignment when all sequences in the cluster are considered. We apply a machine learning based quality control step that uses global cluster statistics as input features to remove poorly clustered gene groupings [37, 38]. While this approach helps with current phylogenetic tree inference algorithms we look towards advancing long read sequencing technologies that these current methods cannot use.

We explore in Chapter 4 the Colored de Bruijn Graph (CdBG) data structure for whole genome phylogenetic tree inference [89]. By using whole, assembled genomes we avoid having to do gene sequence identification as well as paralogous genes. Bubble data structures are identified and sequence alignment is conducted to produce a supermatrix used for tree reconstruction. We find that this method removes the structure (organization) of the genome and produces a black box representation of input organisms which makes drawing biological conclusions about organisms difficult.

We look to building a new data structure in Chapter 5 that is similar to the CdBG but preserves the global structure of the genome [39]. We call this data structure the relaxed de Bruijn Graph and find that it can be used to identify single nucleotide polymorphisms (SNPS) as well as structural variants (translocations and inversions).

The previous work is then augmented in Chapter 6 by formalizing the graph and bubble finding techniques as well as an optimization of the data structure construction algorithm [42].

The main difficulty we then face with the PolyGraph data structure implementation is the slow k-mer indexing operation that must take place. To remedy this, in Chapter 7 we develop a novel and parallel k-mer storage data structure that is superior to all other currently available data structures for k-mer storage [43]. With the new k-mer indexing data structure we are now able to generate the Polygraph quicker and on larger, eukaryote sized genomes.
Figure 1.1: The relative efficiency and resolution of different comparative genomics techniques with our proposed method, the Polygraph, emphasized.

We conclude this work by using the PolyGraph for phylogenetic network and tree inference in Chapter 8. We use the PolyGraph to produce evolutionary splits (ESs) that are biologically relevant and traceable to genomic locations. We find that these evolutionary splits are superior to splits generated from CdBGs (Figure 1.1). Additionally, this framework provides a method for identifying a range of different types of ESs that can be used for phylogeny inference that is not a black box featurization method.
Chapter 2

GNUMAP 4.0: Space and Time Efficient NGS Read Mapping Using the FM-Index

This chapter was published the journal Insights of Bioinformatics [41].

2.1 Abstract

In this article, we present GNUMAP (Genomic Next-generation Universal MAPper), a next-generation sequence read mapper, that utilizes an FM-index as the reference genome data structure. Using the FM-index, GNUMAP is able to map reads with a dramatic decrease in memory usage while maintaining the same mapping characteristics of previous versions. GNUMAP is now able to map NGS reads using a variable kmer size without having to rebuild the reference genome index. Other enhancements were also made to reduce the number of costly Needleman-Wunsch alignments during mapping. This article shows the general benefits of using the FM-index as a reference genome index data structure versus GNUMAP’s previous data structure, the hashmap. Additionally, we show that while the FM-index is sometimes slower than the hashmap, parameter tuning increases mapping speeds significantly, uses less memory and maintains the original mapping characteristics when compared to the hashmap. The new version of GNUMAP is available at https://github.com/byucsl/gnumap

2.2 Background

Next-generation sequencing (NGS) read mapping continues to be an important process in many "-omic" analysis pipelines. The ability to map NGS reads with speed and accuracy has been addressed by many algorithms including Bowtie2 [79], BWA [82], GNUMAP [18] and SOAP [85]. The
Genomic Next-generation Universal MAPper (GNUMAP) is a mapping algorithm that differentiates itself from other mappers by using a probabilistic Needleman-Wunsch (NW) alignment algorithm as well as calculating a posterior probability score for multi-mapped reads [18]. The probabilistic Needleman-Wunsch is unique because it is able to probabilistically align reads to the reference genome by using the raw Solexa/Illumina intensity or probability files. This results in increased confidence in mapping because all possible bases at a particular position are aligned with base-specific uncertainty taken into account. The probabilistic read scoring approach used by GNUMAP is designed to map reads from repeat regions and reduce discarded reads. While the probabilistic mapping processes in GNUMAP has benefits, GNUMAP has remained difficult to use. The data structure used to store the indexed reference genome used for mapping can be too large to fit into memory on many machines, requiring nearly 40GB RAM for the human reference genome for certain kmer sizes.

GNUMAP previously relied on a hashmap to index a reference genome like other NGS read mappers such as SeqMap [68], RMAP [119], and ELAND (Cox, unpublished software). We denote GNUMAP using the hashmap data structure as GNUMAP-hashmap. The hashmap is built by using the kmer as the key in the map. The value for a particular key consists of the genomic coordinates of all occurrences of that kmer in the reference genome. As the hashmap is being built, a maximum threshold (the maximum kmer occurrence) is used to remove kmers that occur too frequently in the genome. This reduces the size of the index and speeds up mapping as kmers that occur more frequently throughout the genome cause frequent false-positive mapping attempts.

The hashmap data structure provides the benefit of quickly looking up genomic coordinates for kmers. It is, however, also limited in several ways. The main limitations are the amount of memory required to hold the entire hashmap in memory and the inability to reuse the hashmap for different length kmers. The hashmap could become extremely large when used with long kmers on a large reference genome. It also needs to be rebuilt for different sized kmers as the hashmap is built with a specific and immutable kmer size. The hash is also limited because it cannot be constructed for kmers where \( k > 15 \). This limits the usable kmer sizes that can be used during
mapping. A variant of GNUMAP uses the C++ Standard Template Library (GNUMAP-STL) to hash the genome to overcome the maximum kmer size limit. To overcome these weaknesses and retain the strengths afforded by probabilistic read mapping, GNUMAP has been modified to make use of an FM-index as the data structure used for indexing a reference genome.

The FM-index is a text compression algorithm that allows for fast substring look-ups in the indexed text [34]. There are many mappers that exist that use this same data structure such as BWA [82]. In this work, we show the effects of implementing the FM-index from BWA in GNUMAP (referenced as GNUMAP-FM). We will show that run-time only suffers moderately while the amount of memory used is greatly reduced allowing GNUMAP to run on commodity desktop hardware with arbitrary kmer sizes. Using the FM-index, GNUMAP is also enhanced by not needing to create multiple reference genome indices for different length kmers. The FM-index also allows for kmers of arbitrary length to be searched for, thus overcoming the kmer size limit imposed by the hash data structure. While changing the underlying data structure used for fast kmer look-ups for read mapping, GNUMAP is still able to maintain the same mapping probabilistic properties from previous versions.

2.3 Methods

Enhancements to GNUMAP include: the implementation of an FM-index for reference genome indexing, added parameters for mappings based solely on kmer look-ups, reduced NW alignments by removing alignment re-computations and other bug fixes. The FM-index and other optimizations were made to GNUMAP version 3.0.2. We release this update as GNUMAP version 4.0.

2.3.1 FM-Index Implementation

The FM-index as implemented in BWA [10] was used as a drop-in replacement for the hashmap used originally in GNUMAP. Users should note that the FM-index in BWA replaces ambiguous bases in the reference sequence randomly with an A, C, G or T. By replacing ambiguous characters, the alphabet of the indexed genome is reduced to four characters and enables 2-bit encoding of
the genome that is more space efficient than using 3-bits or storing the genome as plain text. This randomness added during reference genome indexing causes GNUMAP to produce different results when using the same parameters between the FM-index and hashmap versions. While this is problematic, we find that these issues are only problematic when using small kmers \((k < 15)\) because the shorter the kmer the more likely that a particular kmer is randomly found after the ambiguous base substitution. When using larger kmers \((k \geq 15)\) and additional parameter tuning we see the mapping run-times decrease and number of reads mapped increase drastically (see Table 2.2 for benchmarks).

### 2.3.2 Additional Optimizations and Bug Fixes

In addition to using the FM-index, two other modifications and other bug fixes were made to GNUMAP. First, we implemented a method for mapping reads without using the probabilistic Needleman-Wunsch that is based solely on genomic kmer location look-ups. Second, we modified the mapping algorithm to reduce the number of NW alignments that occur during mapping. Performing the NW alignment is the most costly operation done in GNUMAP. We found that previous versions of GNUMAP would often recompute alignments it had already done. Fixing this resulted in much fewer alignments and much quicker run-times. These modifications, bug fixes and the indexing behavior of the BWA FM-index cause mapping results to vary between the current and previous version of GNUMAP.

### 2.3.3 Datasets

We compared the versions of GNUMAP on how quickly the reads were aligned, how much memory was used to align the reads, and how accurate the alignment was. Four different datasets were used, three simulated and one real dataset.
<table>
<thead>
<tr>
<th>$k$</th>
<th>Hash</th>
<th>FM-Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2.930</td>
<td>7.314</td>
</tr>
<tr>
<td>8</td>
<td>2.936</td>
<td>7.315</td>
</tr>
<tr>
<td>9</td>
<td>3.086</td>
<td>7.326</td>
</tr>
<tr>
<td>10</td>
<td>4.146</td>
<td>7.342</td>
</tr>
<tr>
<td>11</td>
<td>10.47</td>
<td>7.349</td>
</tr>
<tr>
<td>12</td>
<td>19.88</td>
<td>7.341</td>
</tr>
<tr>
<td>13</td>
<td>22.72</td>
<td>7.367</td>
</tr>
<tr>
<td>14</td>
<td>26.27</td>
<td>7.367</td>
</tr>
<tr>
<td>15</td>
<td>38.55</td>
<td>7.350</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>7.335</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>7.341</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>7.346</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>7.346</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>7.350</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>7.345</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>7.334</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>7.345</td>
</tr>
</tbody>
</table>

Figure 2.1: The peak memory usage (in gigabytes) of GNUMap using either the hash or FM-index data structure with a particular kmer size $k$ during run-time. As can be seen, the RAM usage with the hash version increases as $k$ increases while the FM-index implementation remains the same across all sizes $k$. There are no hash results for $k > 15$ due to limitations of the hashing algorithm that does not allow for hashes to be built with $k > 15$.

**WgSim Dataset**

A smaller simulated dataset was generated using WgSim [81]. This smaller dataset consists of 50,000 synthetic 100bp NGS reads from the human reference genome hg19 and was used exclusively for memory profiling benchmarks.

**ART Parameter Tuning**

This dataset consists of 28,880 synthetic 100bp reads generated from the human reference genome hg19 using ART [61]. Using this dataset, we tested the mapping performance of GNUMap with the FM-index. This dataset was generated to ensure we have reads sampled uniformly across the whole genome to see general mapping performance.
ART Mapping Performance

We used the read simulation program ART [61] to generate synthetic 100bp NGS single-end reads. ART generated 14,486,006 reads with 0.5x coverage of the hg19 human genome, with a 0.009% insertion rate and 0.023% deletion rate.

Real Data

For the real data analysis we used a set of NGS reads from the 1000 Genomes Project [21]. We downloaded the low coverage whole genome sequence reads (ERR251013), 65,820,186 total, from The International Genome Sample Resource (IGSR) for the individual HG00140.

All datasets were aligned to the hg19 human genome assembly.

2.3.4 Benchmarking

Measuring Memory Usage

GNUMAP’s memory consumption using the hash and the FM-index was profiled using Valgrind’s heap profiler tool Massif [99]. Using Massif causes the profiled program to slow down tremendously. To profile memory usage, we used the smaller WgSim dataset mapped to the human reference genome hg19. This allowed for memory profiling in a reasonable amount of time.

Parameter Tuning

Parameter tuning with the FM-index shows that results as good, if not better, than GNUMAP-hashmap were possible. Not only were mapping time and percent reads mapped measured, but also how accurate the mappings were. We measured accuracy using the synthetic ART Parameter Tuning dataset.

Mapping results were verified against the ground truth during parameter tuning. We distinguish two types of mapping accuracy: all read accuracy and mapped read accuracy. All read accuracy considers all reads in the readset (mapped and unmapped). Only reads that mapped
uniquely and to the correct location were counted as correct. Reads mapped to multiple locations (multi-mapped read) were counted as mapping incorrectly as well as any read that was left unmapped.

*Mapped read accuracy* is where we look only at mapped reads in the readset. Any read that is unmapped is not counted against the accuracy. Here also, only reads that map uniquely to the correct location are counted as correct. Mapped reads that do not map to the correct location or are multi-mapped reads are counted as incorrect.

### 2.4 Results and Discussion

The main metrics used to test the performance differences between the hash and FM-index versions of GNUMAP were mapping time and memory usage during mapping. Results were generated using various kmer sizes. For the FM-index, we tested using kmer sizes $7 \leq k \leq 23$. We used the original GNUMAP hashing algorithm (GNUMAP-hashmap) for kmer sizes $7 \leq k \leq 15$ and attempted to generate hashmap indices for $k > 15$. All plots were generated using matplotlib [63].

#### 2.4.1 Memory Usage

Memory usage during run-time was measured and can be seen in Figure 2.1. All parameters were maintained between the hashmap and the FM-index runs. Memory usage remains constant with all kmer sizes for the FM-index while the hashmap memory usage grows dramatically. Indexing a genome using the hashmap algorithm and $k > 15$ is not possible unless using the STL variant of GNUMAP.

GNUMAP-FM consumes a great deal less memory when running than the hashmap version. The hashmap index memory usage explodes in size as kmer size grows. The amount of memory required for the hashmap to run using a hash of the human reference genome and $k = 15$ is nearly 40GB. We were unable to generate hashes with $k > 15$. This limitation poses a problem for read mapping because GNUMAP is unable to leverage longer kmers that increase read mapping speed. Because the FM-index uses a constant amount of memory at any size $k$ any kmer size is a usable
mapping parameter that can be tuned to maximize the number of mapped reads. The amount of memory used by the FM-index version allows GNUMAP to be run on even commodity desktop computers with all parameters available where the hashmap version is only able to run on higher-end machines.
<table>
<thead>
<tr>
<th>k</th>
<th>Hashmap Run-Time (HH:MM)</th>
<th>Hashmap Mapped Reads</th>
<th>FM-Index Run-Time (HH:MM)</th>
<th>FM-Index Mapped Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>00:52</td>
<td>0%</td>
<td>00:06</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>00:53</td>
<td>0%</td>
<td>00:06</td>
<td>0%</td>
</tr>
<tr>
<td>9</td>
<td>01:01</td>
<td>0.17%</td>
<td>00:18</td>
<td>0.02%</td>
</tr>
<tr>
<td>10</td>
<td>01:30</td>
<td>1.89%</td>
<td>01:34</td>
<td>1.63%</td>
</tr>
<tr>
<td>11</td>
<td>10:08</td>
<td>46.51%</td>
<td>07:03</td>
<td>44.76%</td>
</tr>
<tr>
<td>12</td>
<td>07:14</td>
<td>70.68%</td>
<td>05:10</td>
<td>70.57%</td>
</tr>
<tr>
<td>13</td>
<td>03:38</td>
<td>69.82%</td>
<td>02:08</td>
<td>69.81%</td>
</tr>
<tr>
<td>14</td>
<td>01:19</td>
<td>65.65%</td>
<td>00:55</td>
<td>65.65%</td>
</tr>
<tr>
<td>15</td>
<td>01:09</td>
<td>59.50%</td>
<td>00:30</td>
<td>59.50%</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>00:21</td>
<td>55.26%</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>-</td>
<td>00:17</td>
<td>55.21%</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>00:15</td>
<td>55.38%</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>00:14</td>
<td>55.05%</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>00:12</td>
<td>52.59%</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>-</td>
<td>00:11</td>
<td>48.85%</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>-</td>
<td>00:10</td>
<td>44.68%</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>00:10</td>
<td>40.57%</td>
</tr>
</tbody>
</table>

Table 2.1: GNUMAP using hashmap and FM-index comparison using the same mapping parameters. Run-time and number of mapped reads using the hashmap and FM-index versions of GNUMAP on the ART generated synthetic dataset and on the real dataset. As kmer size increases, from 7 to 10, run-time is about the same. For kmer sizes 11 and 12 run-time is much larger (see bolded rows). The run-time increase for these kmers is caused by the composition of hg19 where lots of 11 and 12mers (13mers in real dataset where 11 and 12mers didn’t finish) happen to have frequencies near the chosen threshold (1,000) resulting in more NW alignments that are costly in terms of time. For 13mers and larger kmers, the run-time decreases and eventually settles at a minimum run-time.
Differences in these plots show why the 11mer mapping process takes much longer than using 15mers. The maximum kmer occurrence threshold of 1,000 is marked on the plot, kmers with frequencies above this threshold were not used for mapping done in Table 2.1. Area under the curve shows how much of the index is based on rare or frequent kmers. The 11mer index is skewed to the right but has much more of its area concentrated near the occurrence threshold than the 15mer plot. This results in an index that is composed of many more frequent kmers that can trigger false-positive alignments. The 15mer plot is skewed much more to the right and has little area concentrated near the occurrence threshold. An index with this profile is composed of kmers that occur less often throughout the genome causing less alignments to false-positive locations.

### 2.4.2 Run-Time

A comparison of GNUMAP-hashmap to GNUMAP-FM measuring run-time differences can be seen in Table 2.1. Using the same parameters, GNUMAP-FM is able to outperform the hashmap version.

Despite having a slower indexing data structure, removing duplicate Needleman-Wunsch alignments provides significant speedups. The speedup due to alignment deduplication grows as the dataset becomes larger. This is because GNUMAP spends most of its time performing NW alignments and with more reads there are more alignments that are erroneously recomputed.

Mapping runs on the real dataset for kmer sizes 11 and 12 did not finish within the given walltime (24 hours). On the synthetic dataset, the run-time for the 11 and 12mer runs are much higher than those of other kmer sizes. This indicates that there is some characteristic of the kmer composition of hg19 at these particular sizes that made mapping difficult.

We generated kmer frequency profile plots shown in Figure 2.2 to understand why mapping is so much slower using 11 and 12mers versus higher kmer sizes. Kmer counting and histogram counts were generated with Jellyfish [91]. We compared the distribution of the 11mer (problematic
size) with 15mer (usable size). Kmer sizes smaller than 11 were not considered because their mapping performance is poor (see Table 2.1) and moving from 11mers to 15mers provides a large decrease (nearly 9 hours in the synthetic dataset) in run-time. The x-axis of Figure 2.2 is the kmer frequency, how often a particular kmer occurs in the genome. Moving from left to right on the x-axis corresponds to an increase in number of possible alignment operations. This is because kmers that occur more frequently in the genome result in more locations that GNUMAP must check for possible mapping. Kmers that have lower frequencies (left side of the plot) are better for mapping because they occur in fewer places in the human genome (more unique) and cause fewer alignment operations to occur. Alignments performed at genomic positions triggered by frequently occurring kmers are often false-positive mappings and usually discarded. Despite being discarded, the triggered alignment increases the run-time of GNUMAP because the alignment must be performed and scored to decide whether or not to discard it. Figure 2.2 shows the maximum kmer occurrence threshold at 1,000. The area between the 11mer and 15mer plots signifies why using an index with $k = 11$ is so much slower, there are many more 11mers that have $500 \leq \text{frequency} \leq 1000$ than 15mers. This means that when 11mers are used for indexing hg19, there are significantly more kmers that trigger more costly alignments in GNUMAP and result in a huge slowdown.

Looking at run-time and memory usage together reveal even greater limitations of GNUMAP-hashmap. Table 2.1 shows that execution times of using smaller kmers ($7 \leq k \leq 10$) result in very fast run-times and consume very little memory (< 3GB) but very few reads mapped. When using $13 \leq k \leq 15$, GNUMAP-hashmap runs very quickly and is able to map the majority of reads (especially in the real dataset) but it uses enough memory to preclude some users from running it.

### 2.4.3 Index Size on Disk

GNUMAP-hashmap requires a new index to be built for every kmer size and maximum kmer occurrence. Using a different kmer size $k$ results in different kmer profiles and affects the size of the hashmap that is generated. The sizes of the different hashes stored on disk for the human reference hg19 can be seen in Table 2.3. Going from $k = 10$ to $k = 11$ results in the hash doubling in size.
<table>
<thead>
<tr>
<th>Indexing Algorithm</th>
<th>( k )</th>
<th>Max Kmer Occ</th>
<th>Min Kmer Hits</th>
<th>Kmer Jump Size</th>
<th>All Read Accuracy</th>
<th>Mapped Read Accuracy</th>
<th>Run-Time (seconds)</th>
<th>Percent Reads Mapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashmap</td>
<td>15</td>
<td>1000</td>
<td>2</td>
<td>1</td>
<td>90.33%</td>
<td>96.54%</td>
<td>59.773</td>
<td>93.57%</td>
</tr>
<tr>
<td>FM-index</td>
<td>15</td>
<td>1000</td>
<td>2</td>
<td>1</td>
<td><strong>95.12%</strong></td>
<td>95.79%</td>
<td>197.516</td>
<td><strong>99.30%</strong></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>500</td>
<td>2</td>
<td>1</td>
<td>94.97%</td>
<td>95.79%</td>
<td>140.912</td>
<td>99.14%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>200</td>
<td>2</td>
<td>1</td>
<td>94.64%</td>
<td>95.82%</td>
<td>92.796</td>
<td>98.76%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>100</td>
<td>2</td>
<td>1</td>
<td>94.21%</td>
<td>95.89%</td>
<td>71.178</td>
<td>98.24%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>93.59%</td>
<td>95.90%</td>
<td>56.733</td>
<td>97.59%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>25</td>
<td>2</td>
<td>1</td>
<td>92.73%</td>
<td><strong>96.63%</strong></td>
<td>43.153</td>
<td>96.63%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>90.94%</td>
<td>96.27%</td>
<td>26.073</td>
<td>94.46%</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>94.41%</td>
<td>95.88%</td>
<td>20.874</td>
<td>98.46%</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>93.20%</td>
<td>96.06%</td>
<td>15.712</td>
<td>97.02%</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>93.89%</td>
<td>96.06%</td>
<td><strong>14.335</strong></td>
<td>97.81%</td>
</tr>
</tbody>
</table>

Table 2.2: GNUMAP parameter tuning using the FM-index. The FM-index allows for much more flexibility in regards to parameter tuning for read mapping. Here, we compare the GNUMAP with the hashmap to the FM-index with various parameters tuned. Tuning the parameters in GNUMAP-FM greatly increases the number of reads mapped while decreasing the run-time and amount of memory and maintaining mapping accuracy.

This also occurs when going from \( k = 11 \) to \( k = 12 \). In general, the hash size is expected to roughly double every time kmer length is incremented. The hash, however, only stores kmers that occur within the genome resulting in hashes that don’t double in size at each increment of \( k \). The size of the index will also vary based on the maximum kmer occurrence threshold and the specific genome being indexed. GNUMAP-hashmap was unable to generate indices with \( k > 15 \), GNUMAP-STL can theoretically do so but was unable to for the human reference genome on a system with 128GB RAM. The size of the FM-index is 3.13GB on disk and only needs to be stored once for all sizes \( k \). Storing a different hashmap to index the same information with different parameters is unnecessary when data structures like the FM-index exist.

### 2.4.4 Parameter Tuning

Parameter tuning results are shown in Table 2.2. The Max Kmer Occ parameter is the threshold for the maximum number of times a kmer can occur in the genome to be used for mapping. The
<table>
<thead>
<tr>
<th>$k$</th>
<th>Size on Disk (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1.56</td>
</tr>
<tr>
<td>8</td>
<td>1.57</td>
</tr>
<tr>
<td>9</td>
<td>1.73</td>
</tr>
<tr>
<td>10</td>
<td>2.86</td>
</tr>
<tr>
<td>11</td>
<td>9.58</td>
</tr>
<tr>
<td>12</td>
<td>19.58</td>
</tr>
<tr>
<td>13</td>
<td>22.19</td>
</tr>
<tr>
<td>14</td>
<td>23.90</td>
</tr>
<tr>
<td>15</td>
<td>26.93</td>
</tr>
</tbody>
</table>

Table 2.3: GNUMAP hashmap size on disk for various kmer sizes and maximum kmer occurrence 1,000. Hashes with $k > 15$ were unable to be generated.

Min Kmer Hits parameter refers to the minimum number of kmer hits a read needs to trigger an NW alignment and be considered for mapping. Kmer Jump Size specifies the minimum distance between kmer hits in a read. Setting the kmer jump size to 1 results in each kmer from a read being looked up for mapping. Changing various parameters in GNUMAP-FM allows for as many, if not more, reads to be mapped as GNUMAP-hashmap in significantly less time.

Using the FM-index also allows for a great amount of flexibility that was not present when using the hashmap. When using the hashmap version, creating the genome index committed the user to the specific parameters used to build the index, there was no flexibility after it had been built and required a full rebuild if kmer size or maximum kmer occurrence were to be respecified.

Using the FM-index allows us to change these parameters to suit our needs from one sequencing run to another as the genomic composition of different samples can be different. Tables 2.1 and the first two rows of 2.2 show that using the FM-index can cause a dramatic increase in run-time (nearly 4x longer) when identical parameters are used. However, by being able to change the size of $k$ and the maximum kmer occurrence parameters (the two parameters that are immutable when initially creating the hashmap) we find that we can map more reads than the hashmap version in significantly reduced time. While using the FM-index with the same parameters causes a near 4x slowdown in run-time when compared to the hashmap, tuning the parameters causes a near 4x speedup while mapping more reads with higher all read accuracy and comparable mapped read accuracy. This huge speedup was afforded by our ability to choose a kmer size that is impossible
when using either hashmap versions on a genome as large as the human genome. Specifically, setting $k = 23$ and the max kmer occurrence to 10, we were able to map more reads in less than 15 seconds versus the hashmap run which took nearly a minute.

Being able to dynamically change the maximum kmer occurrence also allows us to loosen or tighten our mapping criteria. Using a lower threshold allows for rarer kmers to indicate where we should map reads and decreases extraneous alignments that occur when a common kmer is used for mapping. With the same parameters, the FM-index had a significant slowdown and slight decreases in mapped read accuracy where a higher percentage of mapped reads from the hashmap mapped correctly compared to the FM-index. Using the flexibility of the FM-index, we find that the slight decreases in mapped read accuracy are offset because we’re able to map more of the total readset resulting in an increased overall all read accuracy. We find that through this process we’re not only able to map more reads about as accurately as the hashmap but we can do so in less time using less memory.

The benefits of flexible parameter tuning generalize even further when considering other reference genomes and different quality NGS reads. We found that setting a maximum kmer occurrence of 1,000 was not necessarily the best parameter when mapping to hg19. GNUMAP-hashmap was fixed to its initial parameter choices unless we decided to regenerate the entire hashmap. GNUMAP-FM could change all parameters on demand. When indexing other genomes, it may become apparent that when mapping reads that a higher or lower threshold for maximum kmer occurrence is necessary. This could be the case if using a genome that has many kmers that repeat throughout the genome with relatively fewer unique kmers.

Changing the size of the kmer is also very powerful especially when dealing with NGS reads of lower quality. This becomes extremely pertinent as new sequencing technologies, such as PacBio, generate extremely long reads ($> 500bp$) and have error rates $> 10\%$ that are distributed relatively evenly throughout the read when compared to technologies like Illumina where errors are more likely to occur at the end of the read [11, 98, 108]. Mapping reads with error profiles similar to
PacBio can necessitate using shorter kmers and increasing the maximum kmer occurrence threshold because shorter kmers are more likely to occur frequently throughout a genome than longer kmers.

Changing kmer size and max kmer occurrence thresholds is also beneficial when mapping reads from a specimen that is divergent from the reference genome being used. In studies where the reference genome is of low quality, as is the case for many non-model organisms, lowering kmer size and increasing the occurrence threshold allows for more ambiguous mapping positions to be evaluated. Source specimens can also be different from the reference in the case of highly polymorphic organisms or where high rates of heterozygosity are found and can affect general analysis and sequence processing [36]. The flexibility offered by GNUMAP-FM allows users to work much more easily with non-model organisms, highly polymorphic species and species with high rates of heterozygosity.

2.5 Conclusion

We have shown the benefits of using the FM-index in place of the hashmap data structure in GNUMAP. Initially, we were concerned about using the FM-index because the data structure is slower than the hashmap. Using the same parameters in GNUMAP-hashmap and GNUMAP-FM, we saw that it can result in a large slowdown in GNUMAP-FM’s run-time. We find, however, that using parameter tuning coupled with the other optimizations and bug fixes overcome the possible run-time issues. These results show the impact that parameter tuning has and how important it is to work with the composition of a specific dataset. Finding optimal mapping parameters automatically would be ideal but must take into account the kmer composition of the indexed reference genome and the type of DNA sequencing used. GNUMAP-hashmap is unable to change parameters without triggering a complete index rebuild while GNUMAP-FM, on the other hand, is able to switch between parameters extremely easily. We are now also able to use kmer sizes that were unavailable to us in either GNUMAP-hashmap or GNUMAP-STL. Our new release of GNUMAP-FM is an improvement over the previous version because it is able to map more reads in less time using less memory with about the same accuracy.
Acknowledgment

The authors would like to acknowledge the support of Dr. Christophe Giraud-Carrier, Nozomu Okuda and the members of the Computational Science Laboratory for their help and support during this project.
Chapter 3

The OGCleaner: Detecting False-Positive Sequence Homology

This chapter was published in top tier journals in bioinformatics: BMC Bioinformatics [37] with an applications note in Bioinformatics [40].

3.1 Abstract

Within bioinformatics, phylogenetics is the study of the evolutionary relationships between different species and organisms. The genetic revolution has caused an explosion in the amount of raw genomic information that is available to scientists for study. While there has been an explosion in available data, analysis methods have lagged behind.

A key task in phylogenetics is identifying homology clusters. Current methods rely on using heuristics based on pairwise sequence comparison to identify homology clusters. We propose the Orthology Group Cleaner (the OGCleaner) as a method to evaluate cluster level verification of putative homology clusters in order to create higher quality phylogenetic tree reconstruction.

3.2 Background

In this work, we present the Orthology Group Cleaner (the OGCleaner) as a method for filtering false-positive homology clusters used during phylogenetic tree reconstruction published in BMC Bioinformatics [37] and released as an application in Bioinformatics [40].

One of the most fundamental questions of modern comparative evolutionary phylogenomics is to identify common (homologous) genes that originated through complex biological mechanisms such as speciation, multiple gene losses/gains, horizontal gene transfers, deep coalescence, etc.
When homologous sequences are identified, they are usually grouped and aligned together to form clusters. Homologous DNA (and those translated to amino acids) sequences can be further subdivided into two major classes: orthologs and paralogs. Orthologs are defined as homologous genes in different species that arose due to speciation events, whereas paralogs have evolved from gene duplications. Moreover, orthologous genes are more likely to exhibit a similar tempo and mode of evolution, thus preserving overall sequence composition and physiological function. Paralogs, instead, tend to follow different evolutionary trajectories leading to subfunctionalization, neofunctionalization or both [44]. Nevertheless this phenomenon, called the ortholog conjecture, is still debatable [28] and requires additional validation since it has been shown that even between closely related species some orthologs can diverge such that they eventually lose common functionality.

The accurate detection of sequence homology and subsequent binning into aforementioned classes is essential for robust reconstruction of evolutionary histories in the form of phylogenetic trees [27]. To date, numerous computational algorithms and statistical methods have been developed to perform orthology/paralogy assignments for genic sequences (for review see [76]). Methodologically these approaches employ heuristic-based or evidence (phylogenetic tree)-based identification strategies, which produces varying frequencies of false-positive or negative results. The majority of heuristic algorithms rely on the principle of Reciprocal Best Hit (RBH, [76]) where BLAST [3] hit scores (e-values) approximate evolutionary similarity between two biological sequences. Further algorithmic augmentations of those heuristics, for instance Markov graph clustering (unsupervised learning) [84], enables the definition of orthologous/paralogous clusters from multiple pairwise comparisons. Despite their relatively low computational complexity, these algorithms have been shown to overestimate the number of putative homologies (i.e., higher rates of false-positive detection compared to evidence-based methods [15]).

In this current era of next-generation sequence data researchers have gained access to tremendous amounts of ”omic” data, including for non-model organisms. Phylogenetic information, including species trees, is very limited, unreliable and/or completely unavailable for some poorly studied taxa, thus evidence-based methods are not directly applicable to infer homology. Ebersberger
et al. [29] first attempted to circumvent this problem by using a novel hybrid approach (HaMStR) for extraction of homologous sequences from EST/RNA-seq data using a profile Hidden Markov Model (pHMM) [30] based on a similarity search coupled with subsequent RBH derived from re-BLASTing against a reference proteome. The innovative feature of their approach is in the utilization of pHMM as an additional evidence for homology. This architecture incorporates characteristics of multiple sequence alignments (MSA) for user pre-defined core orthologs. Then, a HMM search is performed with each individual pHMM using matching criterion applied to find putative orthologs in the proteome of interest. This method, however, has limitations and weaknesses, such as:

- Proteome training sets composed of phylogenetically “meaningful” taxa for construction of core ortholog clusters may not be available
- Identification of informative core ortholog clusters may be somewhat cumbersome due to incomplete and/or low coverage sequencing
- The pHMMs may not contain any relevant compositional or phylogenetic properties about biological sequences that constitute MSA
- Inability to explicitly identify paralogy limits the use of HaMStR for some evolutionary applications

Hence, homologous clusters inferred from various multiple sequences require further validation to improve confidence in orthology/paralogy classification. Here, we propose a unique approach to identify false-positive homologies detected by heuristic methods, for example HaMStR or InParanoid [111], called the Orthology Group Cleaner (the OGCleaner). Our machine learning method uses phylogenetically-guided inferred homologies to identify non-homologous (false-positive) clusters of sequences. This improves the accuracy of heuristic searches, like those that rely on BLAST.
3.3 Methods

The OGCleaner is a machine learning approach to removing low quality, putative homology clusters that have been identified by other methods in order to improve phylogenetic tree reconstruction. To do this, we first construct a ground-truth dataset which consists of known homology and non-homology clusters. We then select attributes for use in training machine learning algorithms. Finally, we apply it to two real datasets. One real data provided by a phylogenetic benchmarking suite. The other, a novel set of species where the phylogenetic tree is not well-established.

3.3.1 Construction of ground-truth training sets

The ground-truth data set is built around data gathered from OrthoDB, one of the most comprehensive collections of putative orthologous relationships predicted from proteomes across a vast taxonomic range [124]. This data is particularly useful for construction of training sets since OrthoDB clusters were detected using a phylogeny-informed approach collated with available functional annotations. Hence, training sets constructed from OrthoDB clusters have the inherent benefit of both an evolutionary and physiological assessment resulting in more precise filtering for false-positive homology.

The key to our method was the development of labeled training sets that were used to train supervised machine learning classifiers. Previously, homology clusters were known and annotated in OrthoDB. There were, however, no annotated clusters that represented non-homology clusters from random alignments. Thus, we created and annotated our own set of non-homology clusters through a generative process. We created these clusters in two different manners: randomly aligned sequences and evolving sequences from the homology clusters. Cluster generation can be seen in Figure 3.1.

We extracted 5,332 homology (H) clusters from the predefined OrthoDB profile called ”single copy in >70% of species” across the entire arthropod phylogeny in the database, and then aligned them. Non-homology (NH) clusters were generated by:
Figure 3.1: Cluster generation workflow. This is the process of generating homology and non-homology clusters for training the machine learning algorithms.

1. Using the alignment of randomly drawn sequences from the totality of the protein sequences with cluster size sampled from Poisson ($\lambda$), where $\lambda=44.3056$ was estimated as the average cluster size of Hs.

2. Evolving the sequences taken from H clusters.

This process of evolving sequences was accomplished by using PAML [127] to generate random binary trees for each sequence within a cluster. The discretized number of terminal branches for each random tree was sampled from a normal distribution with mean 50 and a standard deviation of 15. Within each of the clusters, individual sequences were evolved using their respective randomly generated tree using Seq-Gen [109]. We used WAG + I [125] as the substitution model for the amino acid sequences during the evolving process specifying the number of invariable sites (-i) at 0%, 25% and 50%. Then, to form NH clusters, a single evolved sequence from the terminal branches was selected randomly from each tree.

It is unknown how closely the synthetic NH clusters resemble non-homology clusters that are encountered in actual analyses. By following the previously mentioned process, we believe that the simulated NH clusters are realistic clusters in which the evolved sequences are diverged enough to be considered as non-homologous to each other. During run-time, we see that the OGCleaner does filter homology clusters that result in better phylogenetic trees (see Tables 3.8, 3.9 and 3.10). During benchmarking, the OGCleaner identifies clusters as non-homology and results in better.
Figure 3.2: T-SNE manifold learning dimensionality reduction applied to the true-positive OrthoDB clusters (blue, H) and generated false-positive clusters (green, NH). Instances from both classes overlap and are not easily separable suggesting that the generated clusters are similar to the true-positive clusters.

We attempt to assess how realistic the false-positive clusters are by using T-SNE to visualize the H and NH clusters in two dimensions (see Figure 3.2). As can be seen in the figure, the generated instances overlap many of the true-positive clusters from OrthoDB. This suggests to us that the generated false-positive clusters have features that are similar to the original clusters and are therefore realistic to some degree.

From the H and NH clusters, two different sets of training, validation and testing partitions were formed. The first set (EQUAL) had an equal number of homology, randomly aligned, 0% invariable-site evolved, 25% invariable-site evolved and 50% invariable-site evolved clusters within
the combination of training, validation and testing data sets. The second set (PROP) consisted of 50% of the training set as homology clusters while the remaining half of the training set was composed of equal parts randomly aligned, 0% invariable-site evolved, 25% invariable-site evolved and 50% invariable-site evolved clusters. The combined data sets were then partitioned into training, validation and testing. Initial testing revealed EQUAL and PROP to perform about the same. Thus, we retained only the PROP data set for testing and use PROP in our released implementation. This was done by randomly sampling from the pool of clusters and assigning 80% of the clusters (8,800) to training, 10% (1,100) to validation and the last 10% (1,100) to testing.

### 3.3.2 Attribute selection

Ten different attribute features were selected (Table 3.1) and calculated for individual MSA of putative homology clusters and for training Hs and NHs as well. To identify randomly aligned positions in MSAs, we utilized ALISCORE [96], software based on the principle of parametric Monte Carlo resampling within a sliding window. This approach is more objective and exhibits less conservative behavior contrasted to commonly used non-parametric approaches implemented in GBLOCKS [12, 78]. We expected the number of randomly aligned positions for false-positive homologies to be higher than for true homologs. Additionally, several other simple metrics (the number of sequences forming MSAs, alignment length, total number of gaps, total number of amino acid residues and range defined as the difference between longest and smallest sequences within MSAs) were also derived. Overall, incorporation of these attributes into a training set was used to increase the robustness of the performance of the machine learning algorithm. We also obtained amino acid composition for each sequence from each cluster and binned it into four classes according to physicochemical properties of amino acids (charged, uncharged, hydrophobic and special cases), then compositional dispersion was calculated using an unbiased variance estimator corrected for sequence length. Here we assumed that amino acid composition between closely related sequences would be preserved by analogous weak genome-wide evolutionary constraints [75, 123] and thus have diminished variance.
Table 3.1: All Features that were used in order to train the machine learning algorithm. Each of these features was calculated for each of the clusters Machine learning.

For detection of false-positive homology we utilized different supervised machine learning algorithms in order to learn from the labeled data instances. Supervised machine learning algorithms take in labeled instances of a particular event as input. From these labeled instances, the algorithm can then learn from the features associated with the instance to perform classification on other, unlabeled instances. A number of different algorithms were used in order to find a model that performed well. Initially, we used the Waikato Environment for Knowledge Analysis (WEKA) software [53] for training different supervised machine learning classifiers and for evaluating the test data sets. Eventually, we moved to using scikit-learn for our own software package [104]. A set of models was trained and compared using the arthropod data set (see Section 3.5.1 for additional information).

A number of different machine learning algorithms were evaluated. These algorithms included: neural networks, support vector machines (SVMs), random forest, Naive Bayes, logistic regression, and two meta-classifiers. A total of seven models were trained for the arthropod data set. A meta-classifier uses a combination of machine learning algorithms in tandem to perform classification. The two different meta-classifiers utilized stacking with a neural network as the
meta-classifying algorithm. Stacking takes the output classifications for all other machine learning algorithms as input and then feeds them into another machine learning algorithm. The learning algorithm that is stacked on the others is then trained and learns which machine learning algorithms it should give more credence when performing classification. One of the meta-classifiers incorporated all the previously mentioned learning algorithms (neural network, SVM, random forest, Naive Bayes, and logistic regression). The other meta-classifier used all the previously mentioned learning algorithms except for logistic regression. All parameters for each machine learning algorithm are summarized in Table 3.2.

By testing with WEKA, we determined preliminary performance of each of the different algorithms. For our own software package and further analysis, we decided to not include the meta-classifier w/logistic regression (retaining only the meta-classifier with all algorithms) as there appeared to be no added benefit of two meta-classifiers.

### 3.4 Real Data Set Construction

We tested this approach on two different real data sets: a benchmark data set with well-established phylogenetic trees and novel data set where the phylogenetic tree is not well-established. The novel data set was constructed in the following manner:

#### 3.4.1 Library preparation and RNA-seq

For the experimental data set (OD,S) we used 18 Odonata (dragonflies and damselflies) and 2 Ephemeroptera (mayflies) species. Total RNA was extracted from the eye tissues of each taxon using NucleoSpin RNA II columns (Clontech) and reverse-transcribed into cDNA libraries using the Illumina TruSeq RNA v2 sample preparation kit that both generates and amplifies full-length cDNAs. Prepped Ephemeroptera mRNA libraries were sequenced on an Illumina HiSeq 2000 producing 101 bp paired-end reads by the Microarray and Genomic Analysis Core Facility at the Huntsman Cancer Institute at the University of Utah, Salt Lake City, UT, USA, while all Odonata preps were sequenced on a GAIIx producing 72 bp paired-end reads by the DNA sequencing center.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neural Network</td>
<td>weka.classifiers.functions.MultilayerPerceptron</td>
</tr>
<tr>
<td></td>
<td>-L 0.1 -M 0.05 -N 3000 -V 0 -S 0 -E 40 -Ha</td>
</tr>
<tr>
<td>Support Vector Machine (SVM)</td>
<td>weka.classifiers.functions.SMO -C 1.0 -L 0.001</td>
</tr>
<tr>
<td></td>
<td>-P 1.0E-12 -N 0 -V -I -W 1 -K</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.supportVector.PolyKernel -C</td>
</tr>
<tr>
<td>Random Forest</td>
<td>weka.classifiers.trees.RandomForest -I 10 -K 0 -S 1</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>weka.classifiers.bayes.NaiveBayes</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>weka.classifiers.functions.Logistic -R 1.0E-8 -M -1</td>
</tr>
<tr>
<td></td>
<td>weka.classifiers.meta.Stacking -X 10 -M</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.MultilayerPerceptron</td>
</tr>
<tr>
<td></td>
<td>-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a&quot; -S 1 -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.trees.RandomForest -I 10 -K 0 -S 1&quot;</td>
</tr>
<tr>
<td></td>
<td>-B &quot;weka.classifiers.bayes.NaiveBayes &quot; -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.SMO -C 1.0 -L 0.001</td>
</tr>
<tr>
<td>Meta-Classifier w/o Logistic Regression</td>
<td>-P 1.0E-12 -N 0 -V -I -W 1 -K</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.supportVector.PolyKernel -C</td>
</tr>
<tr>
<td></td>
<td>-C 250007 -E 1.0&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.meta.Stacking -X 10 -M</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.MultilayerPerceptron</td>
</tr>
<tr>
<td></td>
<td>-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a&quot; -S 1 -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.Logistic -R 1.0E-8 -M -1&quot; -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.MultilayerPerceptron -L 0.3</td>
</tr>
<tr>
<td></td>
<td>-M 0.2 -N 500 -V 0 -S 0 -E 20 -H a&quot; -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.trees.RandomForest -I 10 -K 0 -S 1&quot; -B</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.bayes.NaiveBayes &quot;</td>
</tr>
<tr>
<td></td>
<td>-B &quot;weka.classifiers.functions.SMO -C 1.0 -L 0.001</td>
</tr>
<tr>
<td></td>
<td>-P 1.0E-12 -N 0 -V -I -W 1 -K</td>
</tr>
<tr>
<td></td>
<td>&quot;weka.classifiers.functions.supportVector.PolyKernel -C</td>
</tr>
<tr>
<td></td>
<td>-C 250007 -E 1.0&quot;&quot;</td>
</tr>
</tbody>
</table>

Table 3.2: The machine learning parameters used for each of the different algorithms in initial training and testing with WEKA.
<table>
<thead>
<tr>
<th>Library</th>
<th>Reads Before Trimming</th>
<th>Reads After Trimming</th>
<th>N50</th>
<th>Max Contig Length</th>
<th>Min Contig Length</th>
<th># Contigs</th>
<th># Peptides (TransDecoder)</th>
<th>SRA ID (NCBI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OD07_Cordulegaster_maculata</td>
<td>7207518</td>
<td>6819629</td>
<td>983</td>
<td>16508</td>
<td>201</td>
<td>28163</td>
<td>11877</td>
<td>SRR2164542</td>
</tr>
<tr>
<td>OD08_Anax_junius</td>
<td>6267456</td>
<td>5978119</td>
<td>807</td>
<td>9457</td>
<td>201</td>
<td>19987</td>
<td>8519</td>
<td>SRR2164543</td>
</tr>
<tr>
<td>OD10_Hetaerina_americana</td>
<td>6447244</td>
<td>6057899</td>
<td>1141</td>
<td>10815</td>
<td>201</td>
<td>34384</td>
<td>13373</td>
<td>SRR2164551</td>
</tr>
<tr>
<td>OD11_Jschurna_Verticalis</td>
<td>6183018</td>
<td>5790475</td>
<td>879</td>
<td>7894</td>
<td>201</td>
<td>27001</td>
<td>10568</td>
<td>SRR2164552</td>
</tr>
<tr>
<td>OD12_Gomphus_Spicatus</td>
<td>6498099</td>
<td>6273168</td>
<td>1502</td>
<td>11612</td>
<td>201</td>
<td>37936</td>
<td>10611</td>
<td>SRR2157378</td>
</tr>
<tr>
<td>OD13_Nebelenna_Gracilis</td>
<td>5894197</td>
<td>5516510</td>
<td>1027</td>
<td>11774</td>
<td>201</td>
<td>33766</td>
<td>12694</td>
<td>SRR2157379</td>
</tr>
<tr>
<td>OD18_Chromagrion_Conditum</td>
<td>7607629</td>
<td>7189745</td>
<td>1188</td>
<td>8671</td>
<td>201</td>
<td>30453</td>
<td>9256</td>
<td>SRR2157380</td>
</tr>
<tr>
<td>OD25_Stylurus_Spinicornis</td>
<td>6840281</td>
<td>6597769</td>
<td>1249</td>
<td>23861</td>
<td>201</td>
<td>37436</td>
<td>13568</td>
<td>SRR2157381</td>
</tr>
<tr>
<td>OD28_Neurocordulia_Yamaskanensis</td>
<td>6410925</td>
<td>6061801</td>
<td>1261</td>
<td>15978</td>
<td>201</td>
<td>34984</td>
<td>12905</td>
<td>SRR2157382</td>
</tr>
<tr>
<td>OD36_Argia_Fumipennis_Violacea</td>
<td>5955971</td>
<td>5600701</td>
<td>1076</td>
<td>11410</td>
<td>201</td>
<td>35049</td>
<td>12754</td>
<td>SRR2157383</td>
</tr>
<tr>
<td>OD42_Archipetes_Grandis</td>
<td>8736454</td>
<td>8367974</td>
<td>1179</td>
<td>9315</td>
<td>201</td>
<td>34318</td>
<td>12285</td>
<td>SRR2164544</td>
</tr>
<tr>
<td>OD43_Hetaerina_Americana_2</td>
<td>3585483</td>
<td>3411596</td>
<td>738</td>
<td>7343</td>
<td>201</td>
<td>24192</td>
<td>7318</td>
<td>SRR2164545</td>
</tr>
<tr>
<td>OD44_Enallagma_Sp</td>
<td>5646110</td>
<td>5370720</td>
<td>940</td>
<td>6446</td>
<td>201</td>
<td>27135</td>
<td>8085</td>
<td>SRR2157367</td>
</tr>
<tr>
<td>OD45_Libellula_Forensis</td>
<td>5591547</td>
<td>5383248</td>
<td>1125</td>
<td>13425</td>
<td>201</td>
<td>31962</td>
<td>11352</td>
<td>SRR2164546</td>
</tr>
<tr>
<td>OD46_Libellula_Saturnata</td>
<td>5981628</td>
<td>5717528</td>
<td>1397</td>
<td>13986</td>
<td>201</td>
<td>35045</td>
<td>13326</td>
<td>SRR2164547</td>
</tr>
<tr>
<td>OD62_Jschurna_Jastata</td>
<td>10080263</td>
<td>9551907</td>
<td>1777</td>
<td>11200</td>
<td>201</td>
<td>40154</td>
<td>13651</td>
<td>SRR2164548</td>
</tr>
<tr>
<td>OD64_Anax_Junius_2</td>
<td>9657180</td>
<td>9195840</td>
<td>1133</td>
<td>21566</td>
<td>201</td>
<td>30833</td>
<td>13117</td>
<td>SRR2157371</td>
</tr>
<tr>
<td>OD_Jschurna_Cervula</td>
<td>7105927</td>
<td>6702900</td>
<td>1156</td>
<td>15621</td>
<td>201</td>
<td>40741</td>
<td>14253</td>
<td>SRR2157372</td>
</tr>
<tr>
<td>R_E001_Baetis_Sp</td>
<td>16352942</td>
<td>16113853</td>
<td>1786</td>
<td>10772</td>
<td>201</td>
<td>30517</td>
<td>16743</td>
<td>SRR2164549</td>
</tr>
<tr>
<td>R_E006_Epeorus_Sp</td>
<td>13846765</td>
<td>13701079</td>
<td>1303</td>
<td>23453</td>
<td>201</td>
<td>45886</td>
<td>16782</td>
<td>SRR2164550</td>
</tr>
</tbody>
</table>

Table 3.3: Sequence Read Archive statistics and IDs.

at Brigham Young University, Provo, UT, USA. The expected insert sizes were 150 bp and 280 bp respectively. Raw RNA-seq reads were deposited in the National Center for Biotechnology Information (NCBI), Sequence Read Archive (see Table 3.3 for SRA IDs).

### 3.4.2 Read trimming and de novo transcriptome assembly

The read libraries were trimmed using the Mott algorithm implemented in PopOlation [72] with default parameters (minimum read length = 40, quality threshold = 20). For the assembly of the transcriptome contigs we used Trinity [48], currently the most accurate de novo assembler for RNA-seq data [129], under the default parameters.

### 3.4.3 Downstream transcriptome processing

In order to identify putative protein sequences within the Trinity assemblies we used TransDecoder ([http://transdecoder.github.io](http://transdecoder.github.io)), the utility integrated into the comprehensive Trinotate pipeline ([http://trinotate.github.io](http://trinotate.github.io)) that is specifically developed for automatic functional annotation of transcriptomes [50]. TransDecoder identifies the longest open reading frames (ORFs) within each assembled DNA contig, the subset of the longest ORFs is then used to empirically estimate parameters for a Markov model based on hexamer distribution. The reference
null distribution that represents non-coding sequences is constructed by randomizing the composition of these longest contigs. During the next decision step, each longest determined ORF and its 5 other alternative reading frames are tested using the trained Markov model. If the log-likelihood coding/noncoding ratio is positive and is the highest, this putative ORF with the correct reading frame is retained in the protein collection (proteome). For more details about the RNA-seq libraries, assemblies and predicted proteomes see Table 3.3.

### 3.4.4 Construction of Drosophila data set

Ten high quality Drosophila raw RNA-seq data sets (DROSO) were obtained from NCBI (Table 3.4). First we trimmed the reads using PoPoolation [72] and subsampled the read libraries to the size of the smallest (Drosophila biarmipes). Then, two additional data sets corresponding to 50% and 10% of the scaled libraries were constructed by randomly drawing reads from the original full-sized libraries. Finally, de novo transcriptome assembly and protein prediction were conducted as outlined above for these three data sets. These data sets were used to test whether homology clusters derived from low-coverage RNA-seq libraries contain more false-positives.

### 3.4.5 Gene homology inference

To predict probable homology relationships between proteomes we used the heuristic predictor InParanoid/MultiParanoid based on the RBH concept [1, 111]. Among various heuristic-based methods for sequence homology detection, OrthoMCL [84] and InParanoid [111] have been shown

---

### Table 3.4: Drosophila data sets.

<table>
<thead>
<tr>
<th>Drosophila Species</th>
<th>NCBI ID</th>
<th># of bases (in Gb)</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>D. ananassae</td>
<td>SRR166825</td>
<td>13.6</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. biarmipes</td>
<td>SRR346718</td>
<td>6.4</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. ficusphila</td>
<td>SRR346748, SRR346751</td>
<td>12.1</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. mauritana</td>
<td>SRR1560444</td>
<td>7.7</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. melanogaster</td>
<td>SRR1197414</td>
<td>9.6</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. miranda</td>
<td>SRR899848</td>
<td>13</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. mojavensis</td>
<td>SRR166833</td>
<td>11.1</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. pseudoobscura</td>
<td>SRR166829</td>
<td>15.1</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. simulans</td>
<td>SRR166816</td>
<td>17.2</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. virilis</td>
<td>SRR166837</td>
<td>15.1</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
<tr>
<td>D. yakuba</td>
<td>SRR166821</td>
<td>13</td>
<td>Illumina HiSeq 200 PE</td>
</tr>
</tbody>
</table>
to exhibit comparable high specificity and sensitivity scores estimated by Latent Class Analysis [15], so in the present study we exploited InParanoid/MultiParanoid v. 4.1 for the purpose of simplicity in computational implementation. InParanoid initially performs bidirectional BLAST hits (BBHs) between two proteomes to detect BBHs in the pairwise manner. For this step, we set default parameters with the BLOSUM62 protein substitution matrix and bit score cutoff of 40 for all-against-all BLAST search. Next, MultiParanoid forms multi-species groups using the notion of a single-linkage. Due to inefficient MultiParanoid clustering algorithm, we had to perform a transitive closure to compile homology clusters for all species together. Transitive closure is an operation performed on a set of related values. Formally, a set S is transitive if the following condition is true: for all values A, B, and C in S, if A is related to B and B is related to C, then A is related to C. Transitive closure takes a set (transitive or non-transitive) and creates all transitive relationships, if they do not already exist. When a set is already transitive, its transitive closure is identical to itself. In the case of the pairwise relationships produced by InParanoid, we constructed orthologous clusters using the notion of transitive closure, where gene identifiers were the values, and homology was the relationship.

For example, our OD\_S data set consisted of N=20 proteomes, we performed \( N(N-1)/2 = 190 \) pairwise InParanoid queries. A simple transitive closure yielded 13,998 homology clusters for OD\_S. The DROSO data set yielded 20,676, 18,584 and 17,067 homology clusters for 100%, 50% and 10% respectively. Then putative homologous genes were aligned to form individual MSA homology clusters for the subsequent analyses using MAFFT v. 6.864b [71] with the ”-auto” flag that enabled detection of the best alignment strategy between accuracy- and speed-oriented methods.

Additionally, we utilized HaMStR v. 13.2.3 [29] under default parameters to delineate putative orthologous sequences in the OD\_S proteome sets. 5,332 core 1-to-1ortholog clusters of 5 arthropod species (Ixodes scapularis, Daphnia pulex, Rhodnius prolixus, Apis mellifera and Heliconius melpomene) for training pHMM were retrieved from the latest version of OrthoDB [124]. We used Rhodnius prolixus (triatomid bug) as the reference core proteome because this is the closest phylogenetically related species and publically available proteome to the Ephemeroptera/Odonata.
lineage [93]. As previously described, each core ortholog cluster was aligned to create MSA using MAFFT and converted into HMM profile using HMMER v. 3.0 [31]. BBHs against the reference proteome were derived using reciprocal BLAST.

### 3.5 Implementation

The OGCleaner is implemented in python2 and utilizes machine learning algorithms to classify putative homology clusters of amino acid sequences as homology or non-homology clusters. It can be downloaded, along with example data sets, from https://github.com/byucsl/ogcleaner. To train the models, positive and negative examples that represent true-positive and false-positive homology clusters are required. Our overall workflow for generating training data is depicted in Figure 3.1. See Figure 3.3 for additional implementation details and example workflow. True-positive homology clusters are gathered from OrthoDB, a hierarchical catalog of orthologous sequences ([77]). Alternatively, a user can provide their own orthology groups for training.

#### Algorithm 1 False-positive cluster generation

1: `procedure GENERATEFPCLUSTERS`
2: Initialize $C$ by duplicating true-positive clusters
3: for each $c \in C$ do
4: for each $seq \in c$ do
5:   $rtree \leftarrow$ random phylogenetic tree with $n$ leaf nodes
6:   $evolvedSeqs \leftarrow n$ evolved sequences based on $rtree$ and $seq$
7:   $evolvedSeq \leftarrow$ randomly selected sequence from $evolvedSeqs$
8:   $seq \leftarrow evolvedSeq$

False-positive clusters are generated by following Algorithm 1. PAML’s ([127]) evolver-RandomTree module is used to generate random phylogenetic trees (Alg. 1 Op. 5). A randomly generated tree with $n$ leaf nodes and a single sequence from a cluster are provided to Seq-Gen ([109]) which generates $n$ evolved sequences (Alg. 1 Op. 6). A single evolved sequence is randomly chosen from the $n$ evolved sequences (Alg. 1 Op. 7) and takes the place of the original sequence in the cluster (Alg. 1 Op. 8). A newly generated cluster consisting of the evolved sequences is still, by definition, a homology cluster because its sequences were derived from the same ancestral
Table 3.5: Confusion matrix of test instances (1835 total instances) for the neural network model.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th>H</th>
<th>NH</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>860</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>NH</td>
<td>42</td>
<td>920</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Per-class performance of the test-set measured with precision, recall and F1-score with support (number of instances) for each class.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.95</td>
<td>0.99</td>
<td>0.97</td>
<td>873</td>
</tr>
<tr>
<td>NH</td>
<td>0.99</td>
<td>0.95</td>
<td>0.97</td>
<td>962</td>
</tr>
<tr>
<td>avg/total</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>1835</td>
</tr>
</tbody>
</table>

sequences. In our released implementation, the training set is 50% H clusters and 50% NH clusters. The new cluster should, however, have diverged enough from the original ancestral sequences to show characteristics of a false-positive homology cluster.

Both the true-positive and false-positive cluster data sets are then featurized for model training. Clusters are aligned using MAFFT ([70]). Cluster features are then extracted using our own feature extraction scripts and Aliscore ([78], [96]). A full list of the used features is found in Table 3.1. Pandas dataframes are used for data handling ([92]). Using scikit-learn, various models are provided to the user for cluster classification ([104]). A meta-classifier that implements stacking created by the authors is also provided to the user. The default model for users is a neural network which has shown to provide superior results compared to other models (see Figures 3.4 and 3.5 for performance graphs). A confusion matrix showing a breakdown of the neural network’s classification accuracy can be found in Table 3.5 as well as the precision, recall, and F1-score in Table 3.6.

3.5.1 Training Data Set

The training data set was used as input to the machine learning model for parameter selection. For the arthropod data set, 80% of the data were used for training, while 10% of the data was reserved for validation and the last 10% for testing. Machine learning algorithms were utilized to learn from
Figure 3.3: A diagram of the overall workflow of the OGClaner. This figure shows the different steps that were used in developing our machine learning model. Arthropod phylogeny was generated in previous studies and deposited in OrthoDB. These sequences were then gathered from OrthoDB and used as our orthology and paralogy clusters. They were combined with generated non-homology clusters. The combination represents our training data set used to train the machine learning algorithms. The experimental data were assembled with proteins inferred from the assemblies. InParanoid was then used to identify putative homologs. Once putative homologs were identified they were input into the trained machine learning algorithms for classification and subsequent cluster trimming.
the combination of the H and NH clusters in the data set to differentiate the two. A trained model could then be used to classify unlabeled instances as homologous and non-homologous. There were a total of 8,800 instances in the OrthoDB arthropod data set that were used as a training set, 4,378 H and 4,422 NH clusters.

### 3.5.2 Validation Data Set

The validation data sets were used after the model had been trained on the training data set. By using the trained model on the validation set, the efficacy of the model could be seen. 10% of the arthropod data set formed the arthropod validation set. The models trained using the arthropod training set were validated only with the arthropod instances. If the model did not perform adequately on the validation set, different parameters for the machine learning algorithms were modified in an attempt to improve the performance of the models. The re-trained models would then revalidate on their same, respective validation sets. The process was repeated until adequate performance of the learning algorithm was reached. The OrthoDB arthropod validation set consisted of 1,100 instances, 566 H and 534 NH clusters.

### 3.5.3 Testing Data Set

All general steps of our pipeline are summarized in Figure 3.3 using the example of OD_S processing. Testing data sets were used only after all the models were finished being trained and validated. This is to ensure an honest measure of the predictive capacity of the models because the testing data were never used in order to evaluate how our model was built and to modify the models. The last 10% of the arthropod data set was used as the arthropod test set. The arthropod test set from the OrthoDB contained 1,100 instances for both the PROP and Equal data sets. The PROP data set had 555 H and 545 NH clusters. The EQUAL data set had 207 H and 893 NH clusters.
3.5.4 Real Data Set

We tested our filtering process by applying the arthropod classifiers trained on the ground-truth data set to the DROSO and OD₅ data sets. Unlike the testing sets mentioned in the previous section, the ground-truth for these data sets was unknown. We examined the number of clusters filtered and conducted a manual inspection of a subset of the filtered clusters to verify the removal of only false-positive homology clusters. Because there are, to the authors’ knowledge, no other post-processing methods for cluster filtering that exist our approach is novel. The filtering processes that do exist are heuristic-based approaches, such as an e-value cutoff, that are built-in modules of the clustering software. Therefore, for comparison, we only examined the number of clusters filtered from the output of InParanoid and HaMStR.

3.5.5 Miscellaneous Parameters

Seq-Gen Parameters

When creating false-positive homology clusters, Seq-Gen is used to evolve the sequences so that they are no longer homology clusters but are more related than clusters of random sequences. Seq-Gen is set to evolve sequences using WAG +I. Invariable sites vary amongst clusters between 0, 25 and 50% so that there is a variety of sequence conservation amongst clusters during evolving.

3.6 Results and Discussion

Model validation shows how a variety of learning algorithms perform using a bootstrap analysis. Figures 3.4 and 3.5 show the performance of the different models using the training/validation and testing data sets. Comparing the different machine learning algorithms, the multi-layer perceptron (MLP) is the model that performs the best. We varied the size of the training set (from 1% to 100% of training instances). On the train/validation data sets, random forest performs the best (~100% accuracy) but does not perform well on the test data set (~90% accuracy) suggesting that it is overfitting to the data. The models behave differently when given varied amounts of data to train
Figure 3.4: Training data set accuracy.
Figure 3.5: Testing data set accuracy
Figure 3.6: Area under the Receiver Operating Characteristic curve (AUROC) for the neural network model. This shows the trade-off between sensitivity (true-positive rate) and specificity (false-positive rate) by varying classification confidence thresholds. An area under the curve (AUC) of nearly 1 shows that there is little trade-off between sensitivity and specificity.
Table 3.7: Summary of InParanoid and HaMStR cluster filtering. The number of clusters that were kept and removed for the OD_S clusters from InParanoid and HaMStR.

<table>
<thead>
<tr>
<th></th>
<th>Kept</th>
<th>Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>InParanoid</td>
<td>10500</td>
<td>3497</td>
</tr>
<tr>
<td>HaMStR</td>
<td>1231</td>
<td>896</td>
</tr>
</tbody>
</table>

on. All models except for Naive Bayes increased in accuracy as the training data grew. Naive Bayes held constant with ~70% accuracy despite the amount of data provided. Additionally we tested which features were the most meaningful for classification using the MLP model (Figures 3.8 and 3.9). We found that the attribute that had the best predictive power to be Aliscore. We believe that this may be the case because clusters with large amounts of randomly aligned positions probably reflect an NH cluster. Creating a heuristic for this attribute may be possible but not straight forward as suggested by random forest’s inability to leverage this feature to outperform the MLP.

Lower coverage data sets are often used when performing transcriptomic and evolutionary analyses especially on non-model organisms. For instance, in a recent paper [97] the authors inferred a phylogeny of many insect species using relatively small RNA-seq library sizes averaging at ~3Gb compared to Drosophila data sets. We expected the number of false-positive clusters to increase with the decreasing sequencing depth. In order to examine this, three DROSO data sets were tested for the presence of false-positives. Indeed, we found that the number of false-positive homology clusters increased in the subsampled DROSO data sets (15.7%, 17.8% and 29.9% for 100%, 50% and 10% DROSO data sets respectively). These subsampled data sets allowed us to see the results that are common when homology clustering is performed on small libraries. Applying the filtering process to the InParanoid and HaMStR OD_S clusters resulted in many removed clusters (Table 3.7), implying that heuristic-based methods have increased rates of false-positives. The removal of many clusters showed the overall poor quality of many of the putative homology clusters (for comparison between homology and false-positive homology clusters see Figure 3.7). This was expected due to the low quality transcriptome assembly that was caused by sequencing depth in addition to biological factors such as interspecific differential expression. The filtering process preserved higher quality clusters and finished almost instantly resulting in huge time savings when
Figure 3.7: Examples of a high quality homology (a) and false-positive homology (b). All sequences within the homology cluster (a) belong to one protein family (FAM81A1-like protein). The sequence in the false-positive homology cluster indicated by the arrow represents Aprataxin and PNK-like factor whereas other sequences represent tyrosyl-DNA phosphodiesterase.
compared to manually curating the clusters. Overall our method can be applied to filter homology clusters derived from closely related (e.g. Drosophila species) as well as highly diverged taxa (e.g. Odonata species). We also note that the trimming procedure behaves more conservatively with increasingly diverged sequences.

Feature validation shows how each individual feature’s classifying power. Results can be seen in Figures 3.8 and 3.9. The same bootstrap analysis scheme as model validation is used except that each feature is only tested using a neural network. All features had testing set accuracies between 50% (as good as random guessing) and ~65% accuracy. These results suggest that it is a combination of the different features that allow the trained models to predict with high accuracy and that it is not a singular feature providing the majority of classification power to the trained models.

Performance was measured by comparing to OrthoMCL [84] using The Orthology Benchmark [2] for evaluating the accuracy of orthology inference. Benchmarking was done using
Figure 3.9: Testing data set per attribute performance using MLP.
the Quest for Orthologs (QoF) Reference Proteomes v5 (2011-04) data set [45]. This data set is generated from UniProtKB [22] and consists of 66 manually compiled proteomes that are a subset of the UniProt reference proteomes. OrthoMCL was used to generate a base set of orthology clusters. The OGCleaner was then applied to the clusters and provided overall better performance than the original OrthoMCL clusters. Comparison using The Orthology Benchmark (http://orthology.benchmarkservice.org/) are shown in tables 3.8, 3.9, and 3.10. We believe that improvements will be even greater when applied to non-model organisms that have lower-quality proteome assemblies.

Trained models can also be evaluated on held-out test data with known labels. This can be used to verify model performance especially if using your own orthology groups for model training. Visualization of performance is provided by matplotlib [64] and IPython [106].
### Table 3.8: Agreement with Reference Gene Phylogenies

Higher values are better for 'pos. predictive value rate' and 'true-positive rate'. The '-' character represents when OrthoMCL + The OGCleaner provided the best performance under the 'Best Method' columns. Improvements are marked in green and weaker results in red. OrthoMCL was used to generate a base set of orthology clusters. The OGCleaner was then applied to the clusters using the provided pre-trained model filtering out low-quality clusters.

<table>
<thead>
<tr>
<th>OrthoMCL</th>
<th>OrthoMCL + The OGCleaner</th>
<th>Best Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos. predictive value rate</td>
<td>pos. predictive value rate</td>
<td>true-positive rate</td>
</tr>
<tr>
<td>APP</td>
<td>0.7722 ± 0.13087</td>
<td>0.7093 ± 0.13572</td>
</tr>
<tr>
<td>'ASTER'</td>
<td>0.9892 ± 0.0210743</td>
<td>0.8582 ± 0.0392417</td>
</tr>
<tr>
<td>BAMBI</td>
<td>0.79± 0.0716831</td>
<td>0.7477 ± 0.116892</td>
</tr>
</tbody>
</table>
| BAR | 0.9696 ± 0.018317 | 0.9375 ± 0.0354241 | 0.9696 ± 0.018317 | 0.9375 ± 0.0354241 | 0.9921 ± 0.0134393 | 0.7553 ± 0.085801 | AntiB |}

### Table 3.9: Generalized Species Tree Discordance Benchmark

Lower values are better for 'avg RF distance(genetree, speciestree)' and 'avg fraction incorrect trees'. Improvements are marked in green and weaker results in red. OrthoMCL was used to generate a base set of orthology clusters. The OGCleaner was then applied to the clusters using the provided pre-trained model filtering out low-quality clusters.

<table>
<thead>
<tr>
<th>OrthoMCL</th>
<th>OrthoMCL + The OGCleaner</th>
<th>Best Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg RF distance (genetree, speciestree)</td>
<td>avg fraction incorrect trees</td>
<td>avg RF distance (genetree, speciestree)</td>
</tr>
<tr>
<td>Eukaryota 10032</td>
<td>0.3097 ± 0.054428</td>
<td>0.9718 ± 0.054428</td>
</tr>
<tr>
<td>SwissTree 0.7553 ± 0.085801</td>
<td>0.9921 ± 0.0134393</td>
<td>0.7553 ± 0.085801</td>
</tr>
<tr>
<td>POP 0.9282 ± 0.018317</td>
<td>0.9375 ± 0.0354241</td>
<td>0.9921 ± 0.0134393</td>
</tr>
<tr>
<td>TreeFamA 0.7891 ± 0.00444051</td>
<td>0.6933 ± 0.00470478</td>
<td>0.8067 ± 0.0045604</td>
</tr>
<tr>
<td></td>
<td>OrthoMCL</td>
<td>OrthoMCL + The OGCleaner</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td>avg Schlicker</td>
<td># ortholog relations</td>
</tr>
<tr>
<td>Gene Ontology Conservation</td>
<td>0.4895±0.0012</td>
<td>135009</td>
</tr>
<tr>
<td>Enzyme Classification Conservation</td>
<td>0.93±0.0010</td>
<td>121280</td>
</tr>
</tbody>
</table>

Table 3.10: Species Tree Discordance Benchmark. Higher value is better for `avg Schlicker` and `# ortholog relations` is better.
3.7 Conclusion

We have demonstrated a machine learning method released as the Orthology Group Cleaner (the OGCleaner) that can be used to differentiate homology and non-homology clusters based on characteristics of known good and bad clusters. These results can be seen in our trained models’ ability to achieve high classification accuracy on the test data sets as well as by examining the number of clusters that were removed from the experimental OD,S data set. We developed a training set of known good and bad clusters that was previously unavailable and made supervised machine learning impossible. Using a feature set that we developed, we tested various machine learning algorithms and found that when trained on our training data sets that the multi-layer perceptron (neural network) consistently outperformed all other models.

Applications of our method were also seen as we applied them to other data sets. Our method was especially useful when applied to the OD,S data set, by filtering out many clusters as false-positive homology. We showed that our method is effective in settings where non-model organisms are being studied and the transcriptome assembly quality is low primarily due to low coverage sequencing or partial RNA degradation.

This paper has demonstrated the usefulness of machine learning in finding homology clusters by quickly removing low quality clusters without using any additional heuristics. The clusters that are retained can then be used later in higher quality phylogeny reconstruction and/or other analyses of gene evolution. In the future, we aim to explore machine learning approaches to clustering sequences more deeply to produce more refined and reliable homology clusters.
Chapter 4

Whole Genome Phylogenetic Tree Reconstruction Using Colored de Bruijn Graphs

This chapter was published at the 17th IEEE International Conference on BioInformatics and BioEngineering (BIBE2017) [89].

4.1 Abstract

We present *kleuren*, a novel assembly-free method to reconstruct phylogenetic trees using the Colored de Bruijn Graph. *kleuren* works by constructing the Colored de Bruijn Graph and then traversing it, finding bubble structures in the graph that provide phylogenetic signal. The bubbles are then aligned and concatenated to form a supermatrix, from which a phylogenetic tree is inferred. We introduce the algorithms that *kleuren* uses to accomplish this task, and show its performance on reconstructing the phylogenetic tree of 12 *Drosophila* species. *kleuren* reconstructed the established phylogenetic tree accurately and is a viable tool for phylogenetic tree reconstruction using whole genome sequences. Software package available at: https://github.com/Colelyman/kleuren.

4.2 Introduction

Whole genome sequences are readily available and affordable like never before [116] due to the advent of high-throughput Next Generation Sequencing (NGS) which has provided researchers with vast amounts of genomic sequencing data that has transformed the landscape of understanding of genomes. The field of phylogenetics, which discovers the evolutionary relationship between taxa, has been no exception to this transformation. Phylogenetics has responded to the copious amounts of high throughput data with novel alignment-free and assembly-free methods [33, 107]
that are better suited [13] to handle the large amounts of data more efficiently than the traditional alignment-based phylogenetic methods. The traditional approach to phylogenetic tree reconstruction requires a homology search throughout the genomes of the taxa, a Multiple Sequence Alignment (MSA) of the homologs, and a tree construction from the resulting matrix. Each of these steps can be computationally expensive and may introduce many unnecessary assumptions that can be avoided by using an alignment-free and assembly-free method.

Alignment-free and assembly-free methods [9, 14, 54, 128] don’t come without their disadvantages, one of which being that many of these methods abstract away the source of the phylogenetic signal to a method akin to shared kmer-counting. We propose an assembly-free whole genome phylogenetic tree reconstruction method using the Colored de Bruijn Graph (CdBG) [67], a data structure that is commonly used for detecting variation and comparing genomes.

The CdBG is similar to a traditional de Bruijn Graph (dBG) in that the substrings of a certain length, referred to as kmers, of a sequence represent the vertices of the dBG and an edge exists between two vertices if the suffix of the first vertex is the prefix of the second vertex. The CdBG differs from the traditional dBG in that each vertex is associated to an unique color (or set of colors) which could be a differing sample, species, or taxon.

We introduce the kleuren (Dutch for ”colors” in tribute of Nicolaas Govert de Bruijn, the de Bruijn graph’s namesake) software package which implements our methods. kleuren works by finding bubble regions [67, 105] of the CdBG, which are where one or more colors diverge at a node, which act as pseudo-homologous regions between the taxa. The sequence for each taxon in each bubble is then extracted and a MSA is performed, then the MSA’s from each bubble are concatenated to form a supermatrix in which a phylogenetic tree of evolution is constructed.

4.3 Methods

4.3.1 Definitions

Given the alphabet $\Sigma = \{A, C, G, T\}$ which are nucleotide codes, let a dBG $G$, be defined as $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_i, \ldots, v_s\}$ is the set of vertices and where $v_i$ is the $i^{th}$ unique
sequence of length \( k \) of \( G \) and where \( E = \{e_1, e_2, \ldots, e_i, \ldots, e_t\} \) is the set of edges and where \( e_i = (v_i, v_{i+1}) \) is an edge connecting two vertices such that the sequence of \( v_i \) and \( v_{i+1} \) overlap by \((k-1)\) characters. Let a CdBG, \( CG \), be defined as \( CG = \{G_1, G_2, \ldots, G_i, \ldots, G_u\} \) for \( u \) taxa where \( G_i = (V_i, E_i) \) is the dBG of the \( i^{th} \) taxon. We refer to each \( G \in CG \) as a distinct color or taxon.

Furthermore, let a path, \( P = (v_1, \ldots, v_w) \) in \( G_i \) be defined as a sequence of vertices from \( V_i \) such that for all subsequences \((v_j, v_{j+1})\) of \( P \), the edge \((v_j, v_{j+1}) \in E_i \). Let a bubble, \( B \), in \( CG \) be defined as \( B = \{P_1, \ldots, P_z\} \) such that each \( P \in B \) is associated with one or more colors, that the first and last vertices of \( \forall P \in B \) are identical, and that \( 2 \leq z \leq u \) (see Figure 4.1).

Finally, let \( K \) be defined as \( K = \{V_1 \cup V_2 \cup \ldots \cup V_i \cup \ldots \cup V_u\} \) where \( V_i \) is the vertices (or the unique kmers) of the \( i^{th} \) dBG, \( G_i \).

### 4.3.2 Software Architecture

We use the \texttt{dbgfm} software package [17] to construct and represent the dBG’s of the individual taxa. \texttt{kleuren} provides an interface to interact with the individual dBG’s to create a CdBG, where each taxon is considered a color. The \texttt{dbgfm} package uses the FM-Index [34], as a space efficient representation of the dBG.

### 4.3.3 \texttt{kleuren} Algorithms

**Overall Algorithm**

\texttt{kleuren} works by iterating over the superset of vertices, \( K \), and discovering vertices that could form a bubble. A vertex, \( s \), could form a bubble if \( s \) is present in \( c \) or more colors of \( CG \), where \( c \) is set by the user as a command line parameter. Note that the lower that \( c \) is, the more potential bubbles that may be found, but \texttt{kleuren} will take longer to run because more vertices will be considered as the starting vertex of a bubble. Let \( s \) be considered as the starting vertex of the bubble, \( b \); then the end vertex, \( e \), of \( b \) is found (see Section 4.3.3). After the end vertex is found, the path, \( p \), between \( s \) and \( e \) is found for each color in \( CG \) (see Section 4.3.3). This process is repeated until
A. Bubble in a Colored de Bruijn Graph

- **Color 1**: Path: ACTGTG
- **Color 2**: Path: ACTAGGTG
- **Color 3**: Path: ACTAGTG

B. Paths in the Bubble of Each Color

Figure 4.1: **A.** An example of a bubble in a Colored de Bruijn Graph with 3 colors (i.e. 3 taxa), and where $k = 3$. The colors of the vertices represent the following: gray- all colors contain the vertex, purple- Color 2 and Color 3 contain the vertex, yellow- Color 1 contains the vertex, red- Color 2 contains the vertex, and blue- Color 3 contains the vertex. In this example ACT is the startVertex and GTG is the endVertex which are both contained in all of the colors. **B.** The extended paths of each color between the startVertex and endVertex.
Algorithm 2 kleuren Algorithm

1: function KLEUREN(K, CG) 
2: \hspace{1em} bubbles ← [] \hspace{1em} \triangleright bubbles is initialized to an empty list 
3: \hspace{1em} for each \hspace{0.5em} k ∈ K do 
4: \hspace{2em} if \hspace{0.5em} k \ is \ in \ c \ or \ more \ colors \ of \ CG then 
5: \hspace{3em} \hspace{1em} endVertex ← FINDENDVERTEX(k, CG) 
6: \hspace{3em} \hspace{1em} for each color \hspace{0.5em} ∈ \hspace{0.5em} CG do 
7: \hspace{4em} \hspace{1em} path ← EXTENDPATH(k, endVertex, color) 
8: \hspace{4em} \hspace{1em} add path to bubble 
9: \hspace{4em} append bubble to bubbles
10: \hspace{1em} alignments ← [] 
11: \hspace{1em} for each bubble ∈ bubbles do 
12: \hspace{2em} \hspace{1em} alignment ← multiple sequence alignment of each path in bubble 
13: \hspace{2em} \hspace{1em} append alignment to alignments 
14: \hspace{1em} supermatrix ← concatenation of alignments

Each vertex in K has been either considered as a starting vertex of a bubble, or has been visited while extending the path between a starting and ending vertex.

Finding the End Vertex

Algorithm 3 Find End Vertex Function

1: function FINDENDVERTEX(startVertex, CG) 
2: \hspace{1em} endVertex ← “” \hspace{1em} \triangleright endVertex is initialized to an empty string 
3: \hspace{1em} neighbors ← GETNEIGHBORS(startVertex) 
4: \hspace{1em} while !ISEMPTY(neighbors) and !ISEMPTY(endVertex) do 
5: \hspace{2em} \hspace{1em} for each neighbor ∈ neighbors do 
6: \hspace{3em} if \hspace{0.5em} k \ is \ in \ c \ or \ more \ colors \ of \ CG then 
7: \hspace{4em} \hspace{1em} \hspace{1em} endVertex ← neighbor 
8: \hspace{1em} return endVertex

The end vertex is found by traversing the path from the startVertex until a vertex is found that is in at least c colors. The endVertex is then used in the function to extend the path (see Section 4.3.3).
Algorithm 4 Extend the Path Functions

1: function EXTENDPATH(startVertex, endVertex, color, maxDepth)
2:     \texttt{path} ← “ ”
3:     \texttt{visited} ← \{\} \quad \triangleright \textit{visited} is initialized to the empty set
4:     if RECURSIVEPATH(startVertex, endVertex, \texttt{path}, color, visited, 0, maxDepth) then
5:         return \texttt{path}
6:     return \texttt{false}

7: function RECURSIVEPATH(currentVertex, endVertex, \texttt{path}, color, visited, depth, maxDepth)
8:     add currentVertex to visited
9:     if depth \geq maxDepth then
10:         return false
11:     if currentKmer == endKmer then
12:         return true
13:     neighbors ← NEIGHBORS(currentVertex, color)
14:     for each neighbor ∈ neighbors do
15:         if neighbor is in visited then
16:             continue
17:         oldPath ← \texttt{path}
18:         append suffix of currentKmer to \texttt{path}
19:         depth ← depth + 1
20:         if \texttt{!RECURSIVEPATH(neighbor, endVertex, path, color, visited, depth, maxDepth)} then
21:             \texttt{path} ← oldPath
22:         else
23:             return true

54
Extending the Path

The main functions that discover the sequences found in a bubble are the Extend the Path Functions (see Section 4.3.3). To extend the path between the startVertex and endVertex we use a recursive function that traverses the dBG for a color in which every possible path between the startVertex and endVertex is explored up to the maxDepth (provided as a command line parameter by the user). The maxDepth parameter allows the user to specify how thorough kleuren will search for a bubble; the higher the maxDepth the more bubbles that kleuren will potentially find, but the longer kleuren will take because at each depth there are exponentially more potential paths to traverse.

4.3.4 Data Acquisition

To measure the effectiveness of our method we used 12 Drosophila species, obtained from FlyBase [49]. We chose this group of species because there is a thoroughly researched and established phylogenetic tree [52].

4.3.5 Tree Construction and Parameters

We used the DSK software package [112] to count the kmers present in all of the Drosophila species. To find the bubbles, we used the following parameters: \( k = 17 \) (kmer size of 17) and \( c = 12 \) (all colors in the CG were required to contain a vertex in order to search for a bubble starting at that vertex) and ran 32 instances of kleuren concurrently for 4 days to find 3,277 bubbles. When all of the bubbles in the CdBG had been identified, we used MAFFT [70] to perform a MSA for each sequence in every bubble that kleuren identified (see Figure 4.2 A.). Then each MSA was concatenated to form a supermatrix (see Figure 4.2 B.) using Biopython [19]. The phylogenetic tree was then inferred from the supermatrix by Maximum Likelihood using IQ-TREE [100] (see Figure 4.2 C.).

Once the tree was constructed, we used the ETE 3 software package [62] to compare the tree to the established one and Phylo.io [115] to visualize the trees.
A. Multiple Sequence Alignment of the Sequences in Bubble (Figure 4.1)

Color 1 Path: ACT--GTG
Color 2 Path: ACTAGGTG
Color 3 Path: ACTA–GTG

B. Supermatrix of Multiple Sequence Alignments concatenated

C. Phylogenetic Tree

Figure 4.2: A. The MultipleSequence Alignment (MSA) of the sequences from the bubble presented in Figure 4.1. B. The MSA’s from each bubble are concatenated into a supermatrix, from which a phylogenetic tree is constructed. C. The resulting tree from the supermatrix inferred by Maximum Likelihood.

4.3.6 Bubble Assumptions

Our method is based on the assumption that bubbles are representative of homologous regions of the taxa genomes. We propose that this assumption is reliable because it has been shown that dBG’s are a suitable method to align sequences [94, 95, 110], and by identifying the bubbles in the CdBG we find the sections of the graph that contain the most phylogenetic signal.

4.4 Results

kleuren constructed a tree (see Figure 4.3) consistent with the established tree found in [52] (the Robinson-Foulds distance [113] between the two trees is 0). Even though we ran many concurrent instances of kleuren for multiple days (see Section 4.3.5), not all of the kmers in K were explored for potential bubbles; meaning that many more bubbles could be found in this CdBG which would only make the phylogeny more concrete.
Figure 4.3: The phylogenetic tree of 12 *Drosophila* species constructed using *kleuren*. This tree resulted from using a kmer size of 17 and required all species to contain a vertex in order for the algorithm to search for a bubble starting at that vertex; and this tree is consistent with the established tree for these 12 species.
Before this final successful run, there were a number of unsuccessful attempts made to construct the tree. Initial attempts were unsuccessful because $K$ (the super-set of kmers) that \texttt{kleuren} uses to find bubbles was semi-sorted (segments of the file were sorted, but all of the kmers in the file were not in lexicographic order) so the vertices that \texttt{kleuren} used to search for bubbles were skewed towards vertices that were lexicographically first. We remedied this issue by shuffling the order of the kmer file so that there was no lexicographic bias towards the bubbles that \texttt{kleuren} finds.

A previous attempt resulted in a tree that had a 0.44 normalized Robinson-Fould’s distance from the established tree occurred because there were too few bubbles, and therefore there was not enough phylogenetic signal for the correct tree to be constructed. To find more bubbles, we split up the kmer file into parts so that multiple instances of \texttt{kleuren} could find bubbles concurrently. We also discovered that there was a high frequency of adenines (A) (a frequency around 40% in comparison to the other nucleotides) in the final supermatrix that could skew the final tree because nucleotides have differing mutation rates. We thought this bias towards A was due to the fact that in the \texttt{recursivePath} function (see Algorithm 4) the \texttt{neighbors} may be sorted, so the function would traverse the \texttt{neighbor} that started with an A before traversing the other \texttt{neighbors} (see Algorithm 4, line: 18). Similar to the previous sorting problem, we shuffled the order of the \texttt{neighbors} so that the first \texttt{neighbor} that was traversed would not always be lexicographically first. Despite this change, the final supermatrix that produced the true tree still had a bias towards A (see Section 4.6).

4.5 Conclusion

We introduced a novel method of constructing accurate phylogenetic trees using a CdBG. Our method, \texttt{kleuren}, uses whole genome sequences to construct a CdBG representation, then it traverses the CdBG to discover bubble structures which become the basis for phylogenetic signal between taxa and eventually produces a phylogenetic tree.

As the NGS era progresses, whole genome sequences are becoming more prevalent for more non-model organisms, in which phylogenies of these organisms have never been constructed.
kleuren is a viable method to relatively quickly and accurately construct the phylogenies for these newly sequenced organisms.

4.6 Future Work

We plan to optimize kleuren so that it can find more bubbles in a shorter amount of time. We will do this by replacing the underlying data structure for how the CdBG is represented. dbgfm, the current method used to represent the dBG in kleuren, sacrifices time efficiency for memory efficiency by storing the FM-Index entirely on disk, thus slowing down queries into the dBG. When kleuren runs faster, more bubbles will be found, and more phylogenetic signal will be present so that a more accurate tree can be constructed.

We also plan to investigate the reasons for the high abundance of A’s in the supermatrix (see Section 4.4) further, and balance the frequency of nucleotides in the supermatrix.

Furthermore, we would like to look into how kleuren performs when the CdBG is constructed using read sequencing data rather than assembled genomes.

Acknowledgment

This work was funded through the Utah NASA Space Grant Consortium and EPSCoR and through the BYU Graduate Research Fellowship.

The authors would like to thank Kristi Bresciano, Michael Cormier, Justin B. Miller, Brandon Pickett, Nathan Schulzke, and Sage Wright for their thoughts concerning the project. The authors would also like to thank the Fulton Supercomputing Laboratory at Brigham Young University for their work to maintain the super-computer on which these experiments were run.
Chapter 5

Genome Polymorphism Detection Through Relaxed de Bruijn Graph Construction

This chapter was published at the 17th IEEE International Conference on BioInformatics and BioEngineering (BIBE2017) [39].

5.1 Abstract

Comparing genomes to identify polymorphisms is a difficult task, especially beyond single nucleotide polymorphisms. Polymorphism detection is important in disease association studies as well as in phylogenetic tree reconstruction. We present a method for identifying polymorphisms in genomes by using a modified version de Bruijn graphs, data structures widely used in genome assembly from Next-Generation Sequencing. Using our method, we are able to identify polymorphisms that exist within a genome as well as see graph structures that form in the de Bruijn graph for particular types of polymorphisms (translocations, etc.)

5.2 Introduction

Detecting polymorphisms in the genome is an important task for an individual specimen (disease association studies) and for a species as whole (phylogenetic tree reconstruction). Whether it be identifying single nucleotide polymorphisms (SNPs) in an individual compared to a reference genome or comparing different species, identifying polymorphic differences is a difficult task. Methods, however, are usually extremely conservative and only identify simple variation (SNPs, insertions, deletions) leaving more complex variation (translocations, inversions) unexamined.
Genomic variation such as translocations and inversions have been shown to cause many human diseases. Translocations have been shown to be the cause of several different types of cancer, such as Burkitt’s lymphoma [56] and acute promyelocytic leukemia [16]. They have also been shown to be associated with schizophrenia [32]. Studying and identifying different types of genomic polymorphisms could have impact on two very important fields in biology: genome wide association studies as well as phylogenetic tree reconstruction.

5.2.1 Genome Wide Association Studies

Commonly, in genome wide association studies (GWAS), next-generation sequence (NGS) reads are mapped to a reference genome. Differences, commonly SNPs and indels, are then identified from the read mapping results. This method has helped identify and associate many mutations with different diseases.

Read mapping, however, is a difficult task. More than 10% of reads were unmapped when mapping 12.2 million reads to the human genome using the popular Burrows-Wheeler Aligner [82]. Some of the reads will be left unmapped due to errors generated during sequencing. Other reads are left unmapped for unknown reasons. It may be that some unmapped reads vary significantly from the reference genome making read mapping difficult.

Mapped reads represent reads that are similar enough to the reference genome to be mapped with a given set of parameters. Unmapped reads may contain more interesting and novel biological information than mapped reads because these reads diverge enough from the reference genome to remain unmapped. Harnessing unmapped reads enables more thorough analysis of how individuals within a species differ and how genomic rearrangements may affect phenotypes.

5.2.2 Phylogenetic Tree Reconstruction

Phylogenetic tree reconstruction is often completed through comparing homologous gene sequences in a group of species of interest. Identification of homologous genes is a difficult task and is often a conservative process, allowing for only gene sequences that are very similar to be clustered together
This approach is limited because it only allows for comparing gene sequences instead of comparing whole genomes [84]. Comparing the entire genome of one species to another is valuable to see if genomic rearrangements or other structural variations occurred to the genome. Accounting for these genomic variations may serve as a future phylogenetic signal in future phylogenetic tree reconstruction.

5.3 Methods

Our method for utilizing unmapped reads and to compare whole genomes is to construct a relaxed de Bruijn graph that allows for more complex genomic variation to be observable.

5.3.1 Standard de Bruijn Graph

A standard de Bruijn graph is a graph structure that represents the genome of an organism. de Bruijn graphs are usually representative of a single species and are commonly used for genome assembly [82, 88]. Beyond genome assembly, they have also been found to increase the percent mapped reads when mapping reads to a de Bruijn graph versus contigs [87].

In a de Bruijn graph, each node represents a unique kmer. Edges in the graph represent kmer overlaps. The graph is usually constructed from NGS reads where reads are broken into kmers and used to populate the graph (see Figure 5.1).

5.3.2 Relaxed de Bruijn Graph

Our relaxed de Bruijn graph differs from a standard de Bruijn Graph is two major ways:

1. The graph contains sequence information for multiple species
2. Kmers can occur multiple times in the graph

By relaxing these constraints on the de Bruijn graph, we are able to identify interesting genomic variation in a tractable amount of time and space. Conceptually, this method can be
Figure 5.1: The construction of a standard de Bruijn Graph. **A)** The original sequence. **B)** sequence broken into kmers \((k=4)\) showing kmer overlap. **C)** A de Bruijn graph with edges formed from overlapping kmers.

thought of as merging two separate de Bruijn graphs by exploiting uniquely occurring kmers in one sequence as anchor points to merge the graphs.

**Graph Construction**

Our graph construction algorithm is outlined in Algorithm 5, also see Figure 5.2 for a visual representation of the graph construction process. After graph construction, we simplify the graph by collapsing neighboring nodes in a graph where that path through the nodes is unambiguous to form unitigs.

**Implementation**

The NetworkX python package [51] was used for storing and manipulating de Bruijn graphs, Gephi [7] and Graphviz [46] were used for graph visualization.


Algorithm 5 Initial relaxed de Bruijn graph construction.

1: **procedure** `CONSTRUCT(seq, k)`
2: **Input:** DNA sequence `seq`, kmer length `k`
3: **Output:** kmer counts `occs`, index counter `curidx`, relaxed de Bruijn graph `g` kmer-index reverse lookup table `rlookup`
4: `occs ←` occurrences of each kmer
5: `curidx ← 0`
6: `g ←` an empty graph
7: **for each** kmer `kmer` in `seq` **do**
8: `l ←` prefix of `kmer`
9: `lidx ← curidx`
10: `curidx ← curidx + 1`
11: `occs[l] ← occs[l] + 1`
12: `rlookup[l] ← lidx`
13: `r ←` suffix of `kmer`
14: `ridx ← curidx`
15: `curidx ← curidx + 1`
16: `occs[r] ← occs[r] + 1`
17: `rlookup[r] ← ridx`
18: `g.addedge(lidx, ridx)`
19: **return** `occs, curidx, g`

Figure 5.2: Construction method for our relaxed de Bruijn Graph for two reference genome sequences. A and D are two different sequences. B and E represent the sequence broken into kmers and the graph node IDs assigned to each kmer. C is the initial relaxed de Bruijn graph containing only A. Blue nodes are unique kmers and red nodes are non-unique kmers occurring in sequence A. F is the resulting relaxed de Bruijn graph once kmers from sequence D are added. Green nodes and edges are new nodes or edges that were added to the graph. See Algorithm 5 for construction of C and Algorithm 6 for F.
Algorithm 6 Appending new sequences after initial graph construction.

1: **procedure** APPEND($g, occs, curidx, seq, k, rlookup$)
2: **Input:** initialized relaxed de Bruijn graph $g$
   kmer occurrence counter $occs$
   index counter $curidx$
   DNA sequence $seq$
   kmer length $k$
   kmer-index reverse lookup table $rlookup$
3: **for each** kmer $kmer$ in $seq$ **do**
4:   $l \leftarrow$ prefix of $kmer$
5:   **if** $occs[l] == 1$ **then**
6:      $lidx \leftarrow rlookup[l]$
7:   **else**
8:      $lidx \leftarrow curidx$
9:      $curidx \leftarrow curidx + 1$
10: $r \leftarrow$ suffix of $kmer$
11: **if** $occs[r] == 1$ **then**
12:      $ridx \leftarrow rlookup[r]$
13:   **else**
14:      $ridx \leftarrow curidx$
15:      $curidx \leftarrow curidx + 1$
16: $g.addedge(lidx, ridx)$
Figure 5.3: Graph structure formed from a synthetic genome and synthetically generated reads after node simplification to unitigs. Graph was constructed using $k = 31$. Blue nodes are sequence from the reference genome and green are sequences from synthetically generated NGS reads. Node 7191 contains a point mutation, node 7187 contains an insertion, node 7188 contains an inversion and the structure that forms from nodes 7185, 7189, and 7190 represent a translocation.

5.4 Results and Discussion

5.4.1 Mapping Synthetic Reads

We created a synthetic genome (6930 base pairs) from and inserted the following polymorphisms:

1. Mutation (position 200)
2. Insertion (position 300, 9 base pairs long)
3. Inversion (position 400, 75 base pairs long)
4. Translocation (position 1000, 50 base pairs long originating from position 600)

150 base pair reads were simulated using ART Illumina [60] at 10x coverage using default settings with no errors and only in the forward direction. We generated a relaxed de Bruijn graph from the original 6930 bps reference sequence and these reads using $k = 31$. The generated graph after unitig simplification can be seen in Figure 5.3.

In the simplified graph, the mutation, insertion and inversion form simple bubble structures (graph structure where a node has multiple outgoing edges to other nodes that later merge as incoming edges into another node) in the graph while the translocation forms a much more complex structure. In these very ideal conditions (all kmers in the reference sequence are unique, reads with no errors), the generated graph shows structures that could be used to generated a phylogenetic signal or for phenotype association with additional generated graphs from other individuals.
5.4.2 Comparing Real Whole Genomes

We compared two real *Escherichia coli* (strain K12) genomes that are very similar. We used *E. coli* K12/MG1655 (U00096.3) and K12/W3110 (NC_007779.1).

Using $k = 1001$, we generated a relaxed de Bruijn graph shown in Figure 5.4. Even with the extremely large $k$, there are still repeated kmers in the graph. Cycles caused by the repeated kmers can be seen in the graph. The graph constructed from real data where repeats occur is much more convoluted compared to the synthetic graph.

5.5 Conclusion and Future Work

In this work, we have presented a method for constructing a unified, relaxed de Bruijn graph that contains more than one sequence source. The relaxed de Bruijn graph enables identification of graph structures that may be used as a signal for phylogenetic tree reconstruction or for use in association studies for phenotypes.

In the future, we plan to augment our algorithm in several ways: to be sensitive to sequencing errors as well as sequenced reads from the reverse direction, be resilient to repeat kmers, remove uninformative bubbles that form from the graph construction process, and identify complex graph structures that form.
Acknowledgment

This work was funded through the Utah NASA Space Grant Consortium and EPSCoR and through the BYU Graduate Research Fellowship.

The authors would like to thank J. Andrew Jacobsen, Paul M. Bodily, Justin Miller, Brandon Pickett, Sage Wright and Michael Cormier for their help and insight during this project.
Chapter 6

The Polygraph: A Data Structure for Genome Alignment and Variation Detection

This chapter was published at the Bioinformatics and Computational Biology 2019 (BI-COB2019) conference [42].

6.1 Abstract

Comparing whole genomes and finding variation is an important and difficult bioinformatic task. We present the Polygraph, a data structure for reference-free, multiple whole genome alignment that can be used to identify genomic structural variation. This data structure is built from assembled genomes and preserves the genomic structure from the assembly. It avoids the “hairball” graph structure that can occur in other graph methods such as de Bruijn graphs. The Polygraph can easily be visualized and be used for identification of structural variants. We apply the Polygraph to *Escherichia coli* and *Saccharomyces cerevisiae* for finding Structural Variants.

6.2 Introduction

Sequence alignment is one of the most basic tools in bioinformatics. Algorithms for sequence comparison, however, are often limited to short sequences and cannot be applied to whole genome sequences due to computational complexity. Aligning only short sequences captures small, local mutations that occur while leaving large-scale mutations undetected. Complete and accurate whole genome alignment is necessary for understanding evolutionary histories of related organisms.

The genome of an organism can evolve in many ways. Small, local mutations include insertions and deletions (indels) and point substitutions. Large-scale genomic modifications in-
clude structural variants (SVs) such as large (> 50 base pair) indels, inversions, duplications and rearrangements such as translocations. As genomes diverge evolutionarily, genomic regions that are ancestrally linked are called homologous. Genome alignment attempts to identify homologous regions amongst a set of genomes.

Previous work in the area of genome alignment has been limited to pairwise alignment or limited to core-genome identification. Methods such as progressiveMauve and Mugsy rely on all-versus-all progressive alignments when applied to many genomes [4, 24]. Methods such as the Harvest Suite rely on core-genome alignment which is a subset of the genome alignment [122]. Core-genome alignment seeks to find orthologous sequences conserved in all aligned genomes. This process is limiting because an all-or-nothing approach does not allow for relationships that exist between subsets of genomes to appear.

Current algorithms usually align using genome anchoring heuristics based on substring seeds. progressiveMauve and Mugsy are both reference-free genome alignment algorithms that use seed anchors [4, 24]. progressiveMauve relies on local multiple alignments (LMAs) which are maximal unique matches (MUMs) [25] that allow for mismatches and occur in multiple genomes. Mugsy first performs pairwise genome alignment using nucmer [26]. The Harvest Suite’s Parsnp aligns genomes by identifying MUMs using a compressed suffix graph and is designed specifically for microbial genomes. Parsnp does not identify SVs, instead focusing only on identifying core-genome regions. Mugsy and progressiveMauve tend to be conservative in their alignments and miss SVs by preferring a consistent global alignment.

In this work, we present a method for positional homology multiple genome alignment [24] that extends our previous work [39]. Genome alignment is made possible by a graph data structure called the Polygraph (PG) which can house multiple genomes and is constructed in a reference-free manner. This data structure contains vertices where homologous regions of genomes are collapsed and edges can show shared recombination events amongst subsets of genomes. Storing multiple genomes in this format facilitates the discovery genomic features useful in comparative genomic analyses.
We demonstrate the efficacy of genome alignments produced by the PG in detecting inversions, translocations and indels. First, we align two yeast (*Saccharomyces cerevisiae*) genomes to verify previously annotated SVs [101] are identified by the PG. We compare these results to Mugsy, Mauve and the Harvest Suite’s Parsnp module. We then compute the PG for 5 *Escherichia coli* and demonstrate how it can be used to identify conserved regions amongst subsets of genomes. The Polygraph provides a method for storing multiple genomes as a graph that allows for the discovery of structural variants.

### 6.3 Methodology

Initially, the Polygraph is a data structure that represents a rough alignment of multiple genomes. All input genomes are anchored together into vertices of the graph by identifying regions of the genome that are assumed homologous. Merging homologous sequence together into vertices makes accessing homologous regions amongst genomes very easy. Initially, homology is identified by using a special set of k-mers (*shared-unique k-mers*). We call this initial alignment rough because only regions we are highly-confident are homologous are merged together. Forming the initial Polygraph provides additional context for genomic sequence and further informs if we can collapse other regions of the input genomes together further simplifying the graph. Through this process, a graph is formed that contains different structures that can represent different types of polymorphisms. Particulars of the Polygraph are detailed below.

#### 6.3.1 Preliminaries

A Polygraph $P = (V, E, k)$ is a simple (no parallel edges), directed graph with vertices $V$ and edges $E$ with parameter $k$ the k-mer size used during construction for a set of genomes $G$. A vertex $v \in V$ represents homologous sequence from multiple genomes or sequence from a single genome by storing genomic coordinates as well as sequence orientation. The maximum in- and out-degree of a vertex $v$ is $n$, the number of genomes present in the graph. A vertex is *merged* if more than one genome is present in it otherwise it is *unmerged*. We use the following helper functions:
1. \texttt{genomesPresent}(v) returns the set of genomes present in vertex \( v \)

2. \texttt{startPos}(v, H) returns a vector of start positions for genomes \( H \) present in \( v \)

3. \texttt{endPos}(v, H) returns a vector of end positions for genomes \( H \) present in \( v \)

4. \texttt{merged}(H) returns true if all vertices in \( H \) are merged vertices

5. \texttt{unmerged}(H) returns true if all vertices in \( V \) are unmerged vertices

6. \texttt{isChild}(v, u) returns true if \( u \) is an immediate successor vertex to \( v \) and \( v \neq u \)

7. \texttt{children}(v) returns the \( \{ u \mid u \in V, \text{isChild}(v, u) \} \)

8. \texttt{grandChildren}(v) returns \( \{ u \mid u, o \in V, \text{isChild}(v, o), \text{isChild}(o, u), u \neq v \} \)

9. \texttt{isParent}(v, u) returns true if \( u \) is an immediate predecessor to \( v \) and \( v \neq u \)

10. \texttt{parents}(v) returns \( \{ u \mid u \in V, \text{isParent}(v, u) \} \)

11. \texttt{count}(s, H) returns the frequency of k-mer \( s \) in genome \( H \)

12. \texttt{occ}(s, H) = \max_{h \in H} \text{count}(s, h)

13. \texttt{sharedUniqueCount}(v, H) counts the number of shared-unique k-mers that make up the sequence represented in vertex \( v \) for genome \( H \)

The set of edges \( E \) is defined as

\[
E = \{ (u, v) \mid u, v \in V, u \neq v, \text{connected}(u, v) \} \tag{6.1}
\]

\[
\text{connected}(u, v) = \begin{cases} 
\text{True} & H \subseteq \text{genomesPresent}(u) \cap \text{genomesPresent}(v), \\
\text{endPos}(u, H) - \text{startPos}(v, H) - 1 = k & \text{otherwise}
\end{cases} \tag{6.2}
\]

An edge represents the path that one or more genomes takes through the graph. For convenience in traversing the graph, we store identifiers for each genome that traverses an edge in an array.
6.3.2 Shared-Unique k-mers

The first step in Polygraph construction is identifying shared-unique k-mers. *Shared-unique k-mers* are k-mers that occur only once within a subset of two or more of the input genomes. First we define the occurrence function $occ$:

$$
occ(x, y, z) = \begin{cases} 
    True & \text{k-mer } x \text{ only occurs } z \text{ times in the set of genomes } y \\
    False & \text{otherwise} 
\end{cases} \quad (6.3)
$$

Shared-unique k-mers are assumed to be homologous. The set of shared-unique k-mers $S$ is

$$
S = \{ (s, H) \mid H \subseteq G, occ(s, H, 1), \text{cardinality}(H) \geq 2 \} \quad (6.4)
$$

For example, given three genomes $A$, $B$ and $C$, a k-mer $x$ that occurs once in $A$ and once in $B$ but multiple times in $C$ would be considered shared-unique for the genomes $A$ and $B$ but not $C$. K-mers that are not shared-unique are called *common*.

Shared-unique k-mers are similar to the maximal unique matches (MUMs) [25] but are not constrained by having to appear in all species, a shared-unique k-mer may exist in any subset of species. This is powerful because instead of all-or-nothing relationships amongst genomes any sub-grouping is permissible. Genomes are then collapsed together using the shared-unique k-mers as anchor points. The graph is simplified by merging non-branching paths together to form unitigs.

6.3.3 Bubble Removal

We simplify the graph by identifying and collapsing bubble structures in the graph. *Bubbles* in the Polygraph represent regions in genomes where polymorphisms such as single nucleotide variants (SNVs) and insertions and deletions (indels) have occurred. They also occur where there is no sequence divergence due to k-mers not meeting shared-unique properties for merging. Bubbles do not represent polymorphisms such as translocations and inversions. Thus, removing bubbles preserves translocations and inversions as graph structures. SNVs and indels can still be recovered.
Figure 6.1: The Polygraph built for three input genomes. Each vertex contains sequence positions (start:end) ordered from genome 0 (top) to 2 (bottom). A $-1$ entry means a genome is not present in a vertex. Colored vertices indicate where sequence is made entirely of shared-unique k-mers with different colors indicating which genomes are present.
by aligning the sequences that a vertex represents. Removing bubbles contracts successor vertices into a predecessor which results in a vertex that may no longer represent the same amount of sequence for each genome present.

A bubble $b$ in the Polygraph consists of a start vertex $\text{start}$, and end vertex $\text{end}$ and set of middle nodes $M$. The set of bubbles $B$ is

$$B = \{ b \mid \text{end} \in \text{grandChildren}(\text{start}), \ M = \text{children}(\text{start}) \cap \text{parents}(\text{end}) \}$$

To collapse a bubble $b$, all sequence from vertices in $M$ are absorbed into $\text{start}$. All vertices in $M$ are removed from a graph $\text{start}$ and $\text{end}$ are connected by a new edge. The vertex $\text{start}$ may now contain sequences of heterogeneous lengths. After bubbles are collapsed, unitiging is performed to compress the graph.

### 6.3.4 Removing Weak Vertices

After merging genomes together using shared-unique k-mers and removing bubbles, we further simplify the graph by removing weak vertices. **Weak vertices**, are merged vertices which may have been randomly merged due to genomes sharing shared-unique k-mers that are not actually homologous. We calculate vertex support to identify weak vertices in the graph. Support for vertex $v$ where $H$ is the set of all genomes present in $v$ is calculated by

$$\text{support}(v) = \min_{h \in H} \text{sharedUniqueCount}(v, h)$$  \hspace{1cm} (6.6)

Weak vertices are removed by creating a new vertex for each genome in $H$ and creating the appropriate edges.
6.3.5 Reflowing

The Polygraph attempts to maximize the amount of homologous sequence contained within merged vertices. To further identify homologous sequence that may not have been found through the initial search for shared-unique k-mers and bubble collapsing, we reflow the Polygraph. Reflowing attempts to move sequence that is contained in common vertices and have as much of it “flow” into a neighboring merged vertex to increase identified homologous sequence. In practice, we accomplish this by generating separate Polygraphs for subgraphs of the original PG replacing the subgraph with the newly generated PG. This process identifies more homologous sequences because an increased number of shared-unique k-mers will be found when only considering the genomic sequence found in the subgraph.

The set of subgraphs, $R$, suitable for reflowing are identified by a single merged vertex $m$ and set of unmerged vertices $N$ such that

$$R = \{(m, N) \mid \text{merged}({m}), \text{unmerged}(N), \text{neighbors}(m, N)\}$$  \hspace{1cm} (6.7)

$$\text{neighbors}(m, N) = \begin{cases} 
\text{True} & N \subseteq \text{children}(m) \cup \text{parents}(m) \\
\text{False} & \text{otherwise}
\end{cases}$$  \hspace{1cm} (6.8)

Each subgraph $r \in R$ is then sent through the PG algorithm producing a new subgraph. Vertices in $r$ are replaced by the newly created subgraph.

6.4 Results

The Polygraph, Mugsy and progressiveMauve were tested on three data sets. First, two yeast *Saccharomyces cerevisiae* strains: EC1118 Genoscope 2009 and the reference genome S288C were
used to see if annotated SVs could be identified. Next, we applied the polygraph to five *Escherichia coli* genomes and visualized the alignment to demonstrate the PG aligning multiple genomes.

In all cases, we formed a Polygraph for genomes using $k = 90$ and the minimum unitig support for weak vertex removal was set to 540 base pairs (bps). For the yeast data set, PG construction took 26m30s on an Intel Xeon E5-2650v4 @2.20GHz. Mugsy’s and progressiveMauve’s runtimes were fast at 1m02s and 2m05s, respectively, but both failed to identify verified SVs that the PG found. We examine three notable structural variations discovered by Novo et al. in chromosomes VI, XIV and XV [101].

### 6.4.1 Yeast Structural Variants

Novo et al. have documented several structural variations that occur between EC1118 and the reference [101]. They make special note of three large-scale rearrangements that occur in chromosomes VI, XIV and XV. Genomes were downloaded from yeastgenome.org. We apply the Polygraph, Mugsy and progressiveMauve to these genomes to identify structural variants. We also attempted to use Parsnp even though it is designed specifically for microbial genomes but were not able to produce comparable results to the other algorithms when applied to a eukaryotic genome.

#### Chromosome VI

Novo et al. identified three SVs in EC1118 chromosome VI. First, a 38 kilobase (kb) novel insertion in the left arm telomere. Second, a 12kb translocation from chromosome VIII situated between the 38kb novel insertion and the left telomere. Lastly, a 23kb deletion in the left arm with 5kb of the deletion translocated to chromosome X.

Using the Polygraph we were able to successfully identify the 38kb insertion and 12kb translocation. Specifically, we found the 38kb insertion to be 38,836bps and located at EC1118:VI (FN393068.1) 0–38,836. The 12kb translocation was a bit shorter at 11,046bps originating from Ref:VIII 53,9634–55,6754 and inserted into EC1118:VI (FN393068.1) 38,747–49,793 and was visualized in Figure 6.2a using Mauve Viewer with MAFFT [69] to produce gapped align-
Figure 6.2: (a) The Polygraph identifies an inverted translocation from chromosome VIII highlighted in red and an inversion in magenta that (b) progressiveMauve does not identify. Visualized using Mauve Viewer.

ments. A graph visualization of the graph component that contains this SV can be seen in Figure 6.3. Both progressiveMauve and Mugsy capture the large 38kb insertion but both miss the 12kb translocation (progressiveMauve shown in Figure 6.2b).

The 23kb Ref:VI deletion with 5kb translocation into EC1118:X was not found by the Polygraph, progressiveMauve or Mugsy. The PG did find a 5kb translocation from Ref:XIV in EC1118:X at the location the 5kb Ref:VI translocation should be. The 5kb translocation came from Ref:XIV 9,739–14,941 and was inserted into EC1118:X (FN393076.1) at 18,6768–19,1969. Neither Mugsy nor progressiveMauve identified this translocation.

We investigated the translocation further by mapping all gene sequences from the reference to EC1118 with BWA [82]. In the 5kb region where the translocation occurred, we found that there were six genes that mapped: three from Ref:VI and three from Ref:XIV forming three putative homologous gene parings that map to the same position in EC1118:X (Figure 6.4). All six mapped genes had only a handful of polymorphisms compared to the EC1118 sequence.

We then compared the 5kb regions from the genomes through multiple sequence alignment (MSA). They were extracted from:

- EC1118:X 186,768–191,969
Figure 6.3: A portion of the Polygraph for yeast chromosome VI. Vertices store genomic coordinates as well as orientation of sequences. Edges of the graph contain a list of all genomes that traverse that edge to facilitate graph traversal algorithms. Coordinates with $-1$ indicate a genome is not present.

Figure 6.4: Mapping of the three genes from Ref:VI (YFL059W, YFL060C and YFL061W) on the top row and Ref:XIV (YNL333W, YNL334C and YNL335W) on the bottom in IGV [114].

- Ref:VI 7,829–13,038
- Ref:XIV 9,739–14,941

MSA was computed using MAFFT [69]. The most notable difference revealed through the MSA was a three base homopolymer thymine deletion in EC1118:X and Ref:XIV. In total, there were five base positions indicating that the Ref:XIV region is more similar to EC1118:X than Ref:VI is.

While this finding contradicts Novo et al.’s statement that the translocation originates from chromosome VI we find sufficient evidence that further investigation on the origins of the translocation is warranted. Additionally, this analysis would not be possible using Mugsy or progressiveMauve as they did not identify it.
Chromosome XIV

This SV is a 17kb novel insertion into Ref:XIV. We found an 18.6kb insertion from EC1118:XIV (FN393084.1) 0–18,654 at the expected location Ref:XIV 558,235. Both progressiveMauve (18,656bps) and Mugsy (18,133bps) identify this insertion as well.

Chromosome XV

This SV is a 65kb replacement of the last 9.7kb in the right arm of Ref:XV. We identified this insertion from EC1118:XV (FN394216.1) 1,045,161–1,110,477 replacing Ref:XV (NC_001147) 1,081,537–1,091,291.

progressiveMauve misidentifies the 9.7kb deletion as a 18.5kb deletion and finds a 6.7kb translocation from Ref:XVI 14,105–18,180 into EC1118:XV 1,036,531–1,040,665 (FN394216.1).

Mugsy identifies the 65kb insertion but misidentifies the 9.7kb deletion. Where the 9.7kb deletion should be, it finds 8 translocations from chromosomes V (FN393065.1), VI (FN393069.1), XII (FN393079.1), XIII (FN393081.1).
6.4.2 Multiple Genome Alignment

The Polygraph can also be used to align, compare and visualize multiple genomes. We aligned five *Escherichia coli* genomes and have visualized the alignment in Figure 6.5. Visualization is a convenient feature because conserved regions can be easily identified as well as heterozygous regions which is useful for identifying potential sites for phylogenetic analysis.

6.5 Discussion

The Polygraph is able to identify numerous structural variants between the two yeast genomes beyond what Novo et al. as well as progressiveMauve and Mugsy were able to identify. Additionally, the resulting graph is small and traversal algorithms can easily be applied. Visual inspection of the PG is simple with yeast-sized genomes and is also human-decipherable. Deeper analysis is easily accomplished as precise genomic coordinates are displayed for each vertex in the graph indicating putative homologous regions.

Of the three structural variants that were indicated by Novo et al., we were able to identify two without caveat with better results compared to Mugsy and progressiveMauve. The main drawback to this fine-grained analysis is runtime. The Polygraph takes significantly more time to run compared to the other software packages. Because the PG is a new algorithm that employs some parallelism there are still many areas where our code efficiency could be increased.

6.6 Conclusion

In this work we have demonstrated the utility of the Polygraph, a new data structure designed for whole genome comparison and analysis. We have demonstrated the construction and refinement algorithms that can simplify a graph representing two genomes enough to be human-understandable when visualized. We also demonstrated the utility of the Polygraph by applying it to the yeast genome for identifying SVs. We also demonstrated results of the PG when applied to more than
two genomes. While superior results are observed, runtime is much longer than similar packages and requires additional work.

Our results show that the Polygraph is a viable data structure for comparing genomes. New methods for leveraging new data are necessary, especially as sequencing technology improves and genome assemblies for individuals become prevalent. Using the Polygraph, structural variants can be found, visualized and analyzed easily. As the Polygraph is extended to handle more genomes it can be used for whole genome phylogenetic tree reconstruction as well as identify complex genomic variations for disease association studies.
Chapter 7

Kcollections: A Fast and Efficient Library for K-mers

This chapter was accepted at the 19th IEEE International Workshop on High Performance Computational Biology (HICOMB2020).

7.1 Abstract

K-mers form the backbone of many bioinformatic algorithms. They are, however, difficult to store and use efficiently because the number of k-mers increases exponentially as $k$ increases. Many algorithms exist for compressed storage of k-mers but suffer from slow insert times or are probabilistic resulting in false-positive k-mers. Furthermore, k-mer libraries usually specialize in associating specific values with k-mers such as a color in colored de Bruijn Graphs or k-mer count. We present kcollections (https://github.com/masakistan/kcollections), a compressed and parallel data structure designed for k-mers generated from whole, assembled genomes. Kcollections is available for C++ and provides set- and map-like structures as well as a k-mer counting data structure all of which utilize parallel operations designed using a MapReduce paradigm. Additionally, we provide basic Python bindings for rapid prototyping. Kcollections makes developing bioinformatic algorithms simpler by abstracting away the tedious task of storing k-mers.

7.2 Introduction

One of the most basic and ubiquitous concepts in bioinformatics are k-mers: sub-strings of length $k$ generated from a DNA sequence. K-mers are the basis of many key bioinformatic algorithms in genome assembly [8, 74, 86], read mapping [79, 83] and phylogenetic analysis [47, 117]. Despite
their widespread use, data structures for storing k-mers can be slow, probabilistic, memory inefficient or restricted to specific use cases.

Current methods that exist for storing k-mers include the Bloom Filter Trie (BFT) [59], Split Sequence Bloom Trees (SSBT) [120] and Mantis [103]. These algorithms have shortcomings that make them difficult to use in the general case.

The BFT is an exact and efficient k-mer store. Insertion operations are slow and has implementation limitations ($k$ must be a multiple of 9). BFT querying is not thread-safe [57].

The SSBT is fast and has low-memory consumption but is a probabilistic structure resulting in false-positive k-mers at rates as high as 57-67% [103]. Additionally, SSBT stores k-mers in a compressed structure that cannot be iterated over to retrieve the k-mers, only explicit k-mer querying is possible.

Mantis is designed for storing k-mers from short-reads. It is reliant on the k-mer counting program Squeakr [102] which requires FASTQ files as input. Squeakr was designed for short-reads and is unable to process longer sequences such as PacBio reads and assembled genomes. Genome assemblies are becoming more prevalent with cheaper and newer sequencing technology makes bioinformatic methods that are designed for chromosome length sequences necessary.

All previously mentioned algorithms can be used to pair “colors” with k-mers. Colors indicate which genomes a k-mer is present in amongst a set of genomes. The BFT can also pair a 2-bit flag to a k-mer. Beyond these cases, no other data can be paired with k-mers.

We introduce kcollections, a library for efficiently storing and accessing k-mers for genomic analysis. It is memory efficient by compressing k-mers into a trie data structure. Speedup is attained by multi-threading k-mer insertions, serialized k-mer caching when adding whole sequences, and thread-safe querying operations. Insertion and reading operations are separated into distinct phases during use. Kcollections has no dependency on k-mer counting pre-processing of input sequences. Additionally, the data structure is generalized for pairing arbitrarily sized k-mers with arbitrary data types. To the authors knowledge, there is no general purpose library that provides exact k-mer indexing that can be paired with values that is fast, flexible, and memory efficient. We implement
kcollections with a C++ API as well as Python bindings. The Python bindings provide drop-in replacements for dict (Kdict), set (Kset) and Counter (Kcounter).

7.3 Methods

Inspired by the Bloom Filter Trie [59], kcollections stores k-mers in a burst trie data structure [55]. It does not use Bloom Filters and adds multi-threaded insert and thread-safe access operations. The methods section proceeds as follows: k-mer serialization, the contents of a vertex in the trie, the insertion operation, how the insertion operation is parallelized and, lastly, the parallel merging function for map/dictionary values.

7.3.1 K-mer Serialization and Caching

K-mers are bit-packed, a single base is represented by 2 bits. Each base is mapped as follows: A:00, C:01, G:10, T:11. An entire k-mer is stored in an array of 8-bit unsigned integers allowing for 4 bases to be stored in each uint8_t. Bit arrays are populated from right-to-left resulting in the right-most 2 bits corresponding to the left-most base in the k-mer. K-mers are sorted quickly by casting their serialized form to an integer or using memcmp. A sorted list of serialized k-mers is not the same as a lexicographic sorting of the k-mers in nucleotide base-space. See Figure 7.1 for an example of k-mer serialization.

Kcollections can insert single k-mers or entire sequences that it breaks up into k-mers. When inserting a sequence, previously serialized k-mers are cached for faster insertions. Caching is achieved by retaining the last \(k-1\) bases in the previously serialized k-mer by performing a right bit-shift by 2. The current nucleotide base is serialized and inserted into the appropriate position via logical or. Because k-mers are stored in arrays of 8-bit uints, the right-shift operation to remove the first base of a k-mer requires storing the first 2 bits of each uint8_t and inserting them to the end of any preceding uint8_t in the k-mer array. To reduce the number of bit-shifts and carryover values, we cast the k-mer to a 64-bit uint array.
Figure 7.1: K-mer serialization packs each base into 2-bits. The left-most base in a k-mer is stored in the right-most position when serialized. Sorting serialized k-mers is done using C’s `memcmp` function resulting in a non-lexicographic sorting in nucleotide base-space.

### 7.3.2 Vertices

Vertices in the trie each contain an uncompressed container of k-mers, a child vertex array and a child vertex presence array. All vertices except leaves represent a 4 nucleotide segment of DNA that is a prefix for all its children. A 4 base prefix was chosen because k-mers are stored in 8-bit uint arrays. Leaf nodes represent \( k \mod 4 \) bases.

#### Uncompressed Container

As k-mers are inserted, they are initially stored in the root vertex in an “uncompressed container” which is a sorted list of serialized k-mers. K-mers are sorted using the C `memcmp` function. This results in a k-mer sorting that is fast. Once a sufficient number of k-mers have been inserted into the uncompressed container it bursts. When a container in a vertex bursts, child vertices are added to that vertex. K-mers in the uncompressed container with the same 4-base prefix are inserted into the same child vertex. When a new child vertex is created, the child vertex presence array is updated. An index into the array is found by casting the serialized 4-base prefix to an unsigned integer and setting the presence array at that position to 1. This results in an emptied uncompressed container and compressing k-mers that share prefixes into child vertices. After bursting, K-mers are inserted into the uncompressed container unless the input shares a 4-base prefix of one of the child vertices. In that case, we traverse to the child vertex and repeat the process of determining where to insert the k-mer either in the vertex’s uncompressed container or to a child vertex. See Figure 7.3 for an example.
(a) The original child vertex presence array.
(b) The 224 bit shifted array.
(c) The child vertex array.

Figure 7.2: Finding the child vertex that represents a given 4-mer by calculating the hamming weight of the child vertex presence array. The child vertex for k-mer AAGA is found by serializing the k-mer 00100000. Next, the serialized value is cast to an int (32 in this case) and is used as an index into the population array. The count of 1’ss that precedes this index (cells highlighted in orange) correspond to the index into the child vertices array for this k-mer. This count is determined by left-shifting the child vertex presence array (a) by 256 – 32 = 224 bits to produce (b). The hamming weight in the shifted array is found by counting the 1s left in the array. In this case 9 1s correspond to the correct index in the child vertex array (c).

Child Vertex Array

A list of child vertices sorted by the child’s prefix is maintained in each vertex. A vertex has a maximum of 256 child vertices which represent the unique 4-mers. If the child vertex array is not fully populated we compress it by removing any vacancies that exist. We calculate the hamming distance of the child vertex presence array to determine what 4-mer a particular child vertex represents. See Figure 7.2 for identifying a child vertex given a 4-mer prefix.

Child Vertex Presence Array

The child vertex presence array is stored as a 256-bit unsigned int provided by the uint256_t library [80]. In order to determine which child represents a 4-mer in the child vertices array the 4-mer is serialized and cast as an integer index into the presence array. The hamming weight or population count is then calculated for all bits that precede this index and indicates the index in the
child vertex array that corresponds to that 4-mer. If the found index equals the length of the child vertex array there is no child vertex that represents the given 4-mer.

### 7.3.3 Parallel Insertion

Kcollections is a multi-threaded application and performs k-mer insertion using a MapReduce scheme. Kcollections partitions the underlying trie so that a single partition serves a specified ranges of k-mers ensuring no race conditions. Incoming k-mers are serialized, sorted and mapped to the appropriate thread/partition. Once all k-mers are inserted, the threads are joined together and the partitions for each respective thread are reduced into one. The number of partitions/threads must be a power of 2.

#### Partitioning the Trie and Sorting K-mers

Race conditions occur when multiple threads read or write to the same part of the data structure. We solve this by partitioning the trie, assigning each thread to only store k-mers that have a 4-mer prefix that falls within a certain range. The size of each partition is determined by \( \frac{256}{n_{\text{threads}}} \).

For example, if \( n_{\text{threads}} = 4 \), the number of partitions is 4 and each partition covers 64 prefixes. Thread 0 is assigned all prefixes that fall between 0 and 63 when the 4-mer prefix of the serialized k-mer is cast as an unsigned int. The same pattern follows for the other threads.

Sorting and mapping k-mers can be an expensive operation. We overcome this by taking the 8-bit/4-mer prefix of a serialized k-mer, bit shifting it \( s \) bits and casting the resulting value as an unsigned integer \( i \) where \( i \) corresponds to the correct thread id. We determine the number of bits to shift \( s \)

\[
s = 8 - \log_2(t) \tag{7.1}
\]

where \( t \) is the number of threads such that

\[
\{ t = 2^x \mid x \in \mathbb{Z}, 0 \leq x \leq 8 \} \tag{7.2}
\]
Figure 7.3: Fast k-mer sorting. Given $t = 4$ threads, the number of bits to shift is $s = 6$. The appropriate consumer thread is determined quickly by shifting the first 4-mer prefix of the k-mer $s$ bits and casting it as an unsigned integer $i$ which corresponds to the appropriate consumer thread.

This is a fast operation because the k-mers are already serialized and bit shifting and casting are very cheap compared to string comparisons. As an example of how k-mers are sorted and sent to the appropriate thread refer to Figure 7.3.

**Partition Work Queues/Buffers**

Mapped k-mers are stored in a series of buffers assigned to specific partitions. There are $j$ buffers for each of the $w$ partitions. Each buffer holds $c$ k-mers to be inserted. In practice, we found good performance with $j = 10$ and $c = 500$. A buffer is filled completely by mapping $c$ k-mers to it and then signaling to the partition that buffer is ready for storing.

**Reducing Partitions**

Once all k-mers have been inserted, the different partitions must be reduced together for querying and other operations. Each partition acts as a self-contained kcollections structure for k-mers of a certain prefix range. We merge these different structures together by taking each of the child vertex arrays and concatenating them together in sorted order. The concatenated list is set as the child vertex array of the root vertex of the reduced data structure. The child vertex presence array of the root vertex $a_{\text{root}}$ is updated as well

$$a_{\text{root}} = \bigvee_{i=1}^{t} a_i$$  \hspace{1cm} (7.3)

where $t$ is the number of consumer threads defined in Equation 7.2. See Figure 7.4 for an example.
Figure 7.4: After insertion, all consumer threads are merged together. (a) shows child vertex presence arrays for each thread highlighting the 4-mer prefixes that each thread handles. Consumer thread joining concatenates these arrays together using logical or and storing the result in the producer thread. (b) Child vertex arrays are concatenated together as well. Different colors indicate which thread a k-mer prefix was originally assigned to.

7.3.4 Merging Map/Dictionary Values

When a k-mer is present multiple times with different paired values during insertion of a sequence (not a single k-mer) in a map/dictionary it is unclear which value should be stored. This occurs when creating a dictionary that stores the position each k-mer occurs at. Kcollections allows for a merging function to be specified that indicates how different values for the same k-mer should be reconciled.

The merging function is flexible and allows any operation to merge the previously stored and new values. If no merge function is provided, the old value is replaced with the new value. We provide an example in Python (Listing 7.1) and in C++ (Listing 7.2) and with additional examples in our documentation.

7.4 Results and Discussion

We compare the kcollections Kdict module to other algorithms in memory consumption, run-time and correctness when querying. We test against the CPython 3.7.1 built-in set, SSBT and the
Listing 7.1: Merging function example in Python that stores the position of each k-mer occurrence in a list. prev_value and new_value are both lists that initially hold a single genomic position. As the same k-mer is inserted, the positions accumulate.

```
def merge(prev_value, new_value):
    prev_value.append(new_value[0])
    return prev_value
```

Listing 7.2: Merging function example in C++ for k-mer counting. The map is instantiated to store ints as values. When k-mers are inserted they are all inserted with a value of 1. As the same k-mer is inserted, the values are summed together resulting in k-mer counting.

```
int& merge(int& prevVal, int& newVal) {
    prevVal += newVal;
    return prevVal;
}
```

<table>
<thead>
<tr>
<th>Program</th>
<th>Subprogram</th>
<th>Elapsed Wall Clock Time (hh:mm:ss)</th>
<th>Maximum RAM Used (GB)</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSBT</td>
<td>SSBT hashes</td>
<td>00:00:00</td>
<td>0.009</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SSBT count</td>
<td>00:30:21</td>
<td>30.144</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>SSBT build</td>
<td>00:00:00</td>
<td>0.009</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SSBT compress</td>
<td>00:01:05</td>
<td>1.855</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>total</strong></td>
<td>00:31:26</td>
<td>30.144</td>
<td>-</td>
</tr>
<tr>
<td>BFT</td>
<td></td>
<td>03:53:38</td>
<td><strong>16.281</strong></td>
<td>1</td>
</tr>
<tr>
<td>Python Set</td>
<td></td>
<td>02:00:56 (22:20:24)</td>
<td>261.527</td>
<td>1</td>
</tr>
<tr>
<td>kcollections</td>
<td></td>
<td><strong>00:27:11</strong></td>
<td>24.105</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 7.1: Time and memory usage for indexing the human genome. The BFT requires k-mer generation in a pre-process step, we use jellyfish and include those results. Both the build time and overall running time are provided for the Python set due to the large disparity between them.

<table>
<thead>
<tr>
<th>Program</th>
<th>Elapsed Wall Clock Time (mm:ss)</th>
<th>Maximum RAM Used (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFT</td>
<td>01:25</td>
<td>16.166</td>
</tr>
<tr>
<td>Python Set</td>
<td><strong>01:00</strong></td>
<td>261.676</td>
</tr>
<tr>
<td>SSBT</td>
<td>11:17</td>
<td><strong>5.396</strong></td>
</tr>
<tr>
<td>kcollections</td>
<td>01:29</td>
<td>22.840</td>
</tr>
</tbody>
</table>

Table 7.2: Time and memory usage for 20M queries against the human reference genome. 10M k-mers that exist and 10M that do not exist in the index.
BFT paired with jellyfish [90] for k-mer counting. The Python set is included because it is a general purpose data structure that is useful in algorithm development and which kcollections seeks to supplant for k-mer storing. We were unable to test against Mantis due to its pre-processing dependency Squeakr’s inability to process human chromosome length sequences.

Testing was done by indexing the human reference genome GRCh38 with $k = 27$. Querying was tested using 20 million 27-mers: 10 million that do not exist in the human genome and 10 million that do. We used 16 threads in all cases where multi-threading was available using a machine with an Intel Xeon E5-2650 v4 @2.20GHz and 256 GB RAM running Ubuntu Linux 16.04.

### 7.4.1 Memory Usage

In overall memory usage for building the data structure, BFT uses the least memory at 16GB (see Table 7.1). Kcollections next with 24GB. SSBT must be generated in many steps like the BFT with k-mer counting also being the step that requires more memory (30GB). The Python set memory usage results are prohibitively high (over 256GB), maxing out RAM and requiring swap space for a relatively small size $k$ makes it impossible to use on most computers.

For querying memory usage, SSBT had the best performance using only 5GB (see Table 7.2) While this memory usage is much lower than the other data structures it comes at the cost of correctness that none of the other algorithms suffer from. The BFT (16GB) is superior to kcollections (24GB) while the Python set again requires a huge amount of memory that makes it infeasible to use in most situations.

### 7.4.2 Run-time

Kcolletions has the fastest indexing time at 00:27:11 (hh:mm:ss) as can be seen in Table 7.1. SSBT is just slightly slower while BFT takes more than 4 hours to finish. We make note that the Python set takes 02:00:56 to insert all k-mers and creates a very large object. This object causes the overall script run-time to be tremendous (22:20:24) because of an issue Python has with deallocating large objects [20].
All querying was done in serial and with results in Table 7.2. The Python set is the fastest finishing in 1 minute. BFT and kcollections are comparable at 01:25 and 01:29 (mm:ss), respectively. SSBT takes the most time at 11:17.

### 7.4.3 Correctness

The Python set, BFT and kcollections are all correct for the 20 million queries. Because the SSBT is an inexact compression method it results in a false-positive rate of 55.53% which corroborates findings by Pandey et al. [103]. As expected, SSBT reported no false-negatives. The existence of a false-positive rate this high makes the SSBT unusable in some situations and can greatly outweigh its low-memory advantage during querying.

### 7.5 Conclusion

In this article, we presented kcollections, a library of data structures for fast, efficient and general purpose k-mer-based algorithm development. These data structures can be constructed quickly using large DNA sequences with sizes up to whole, assembled genomes or sets of reads. It is flexible because it is not constrained to specific k-mer sizes and it is user friendly because it does not require any pre-processing by other programs. Kcollections is more generalized than other algorithms because it can pair arbitrary data with k-mers instead of only color information or a 2-bit flag. This makes kcollections preferable in general algorithm development when other types of information need to be associated with k-mers. Though it does not perform best in every measure, its overall performance makes it an ideal choice.

### Acknowledgements

The authors would like to thank members of the Computational Sciences Laboratory at BYU.
Funding

This work was funded by the Utah NASA Space Grant Consortium and EPSCoR as well as the BYU Graduate Research Fellowship.
Chapter 8

Whole Genome Phylogenomics Using the PolyGraph

8.1 Introduction

Comparative genomics is an essential task in bioinformatics. Identifying regions of similarity and dissimilarity between different organisms illuminate functional, structural and evolutionary relationships. Many current methods rely on alignment-based methods which make strong assumptions about the linearity and conserved nature of DNA sequences from different species. Alignment-free options are usually much faster and can be more flexible in their assumptions in regards to conserved sequence but usually lose biological relevance due to their method of featurizing a genome.

Whole genome alignment is usually infeasible with current alignment algorithms which are usually $O(n^2)$. In previous works we have introduced the PolyGraph (PG) [42] which is a graph data structure constructed from several different genomes. This data structure is constructed through a pseudo-alignment algorithm that anchors genomes together using shared-unique k-mers. This graph structure can then be examined to identify different types of polymorphisms.

In this work, we use the PolyGraph to identify conserved sequence shared amongst different genomes to produce evolutionary splits (ESs) [6] and infer phylogenetic networks. The phylogenetic network is produced by aggregating the evolutionary splits together using the Symmetric Alignment-free phylogeNomic Splits (SANS) algorithm [126] and then using SplitsTree [65, 66] to compute weakly compatible [5] phylogenetic networks and a bifurcating, compatible phylogenetic tree using scripts from Wittler [126]. The original SANS algorithm was applied to ESs that were generated from the colored de Bruijn Graph (CdBG). We find that the ESs generated by the PG to perform better than those generated by a CdBG. Additionally, introspection of ESs from the PG is possible
since the PG conserves genomic coordinates from its source genomes. Lastly, we show that the benefits of introspection by identifying inversions and using those in conjunction with the SANS algorithm to generate phylogenetic networks.

In the following sections we outline the pitfalls of using a colored de Bruijn graph in generating phylogenetic splits and why the PolyGraph remedies these weaknesses. We give a brief overview of how the initial PolyGraph data structure is constructed from a set of genomes and post-processing steps to simplify the graph and how different evolutionary splits are identified in the graph. We then outline modifications the SANS algorithm for use with ESs mined from the PG. We show the three reconstructed phylogenetic networks for the some of the same species that the original SANS paper examined: Prasinoviruses, *Escherichia coli* and for *Drosophila* using different ESs gathered from the PG data structure. We do not examine all the same datasets because of the PG’s limitation to only 32 genomes.

### 8.2 Background

Colored de Bruijn graphs (CdBGs) can be used in comparative genomics in a manner that is both alignment- and reference-free. Fast construction algorithms that utilize NGS reads (58; 59) make CdBGs a convenient method for identifying phylogenetic signal amongst a set of genomes (126). CdBGs have the drawback of losing context of each inserted k-mer because k-mers of the same sequence are collapsed into a single vertex within the graph, resulting in black box generation of phylogenetic signal that cannot be further examined afterwards. This becomes more problematic as sequencing technology produces longer reads and genome assemblies become more prevalent.

We present a method for generating evolutionary splits using the PolyGraph data structure (42) combined with an extended version of the SANS algorithm (126). The PolyGraph data structure is a graph representation of multiple genomes that collapses homologous regions into single vertices. This structure maintains genomic coordinates after all genomes are input and can be used to generate evolutionary splits in various ways. Here, we explore two methods for identifying evolutionary splits: a vertex-centric method and focusing only on inversions. Vertex-centric splits are related
to conserved sequence between different genomes and use all vertices present in the PG. We also
demonstrate the ability to generate ESs from different types of polymorphisms by performing
analyses with a modified vertex-centric method that examines inversions only. Other approaches
also exist such as using an edge-centric splits which combines sequence conservation with genomic
structure and organization. We leave edge-centric splits for future work.

8.3 Algorithm

We first give a brief overview of the PolyGraph construction algorithm, the extended SANS
algorithm for generating evolutionary splits and, finally, examining evolutionary splits using genomic
coordinates.

8.3.1 The PolyGraph

The PolyGraph is a graph data structure where shared vertices amongst genomes indicate shared/ho-
monologous sequence and edges indicate shared genomic structure. It is similar to the de Bruijn graph
but differs in how vertices are collapsed together. In a de Bruijn Graph, all k-mers of the same
sequence are collapsed together but in the PolyGraph k-mers are identified as shared-unique or not.

First, we define the occurrence function:

\[
occ(x, y, z) = \begin{cases} 
  True & \text{k-mer } x \text{ only occurs } z \text{ times in the set of genomes } y \\
  False & \text{otherwise}
\end{cases}
\]  

To define a shared-unique kmer \( s \) as a k-mer in a set of genomes \( G \) that has the following properties:

\[
s \mid H \subseteq G, occ(s, H, 1), cardinality(H) \geq 2
\]  

Shared-unique k-mers are collapsed into single vertices in the graph. This results in multiple vertices
in the graph possibly representing the same k-mer but allows us to preserve genomic coordinates in
the constructed graph. It is built using the k-mer indexing algorithm kcollections ([43]).
Usually, in a graph structure like this, a bubble structure in the graph is indicative of a polymorphism such as a single nucleotide polymorphism (SNP) or insertion/deletion (indel). In the PG, bubbles can form because k-mers are not shared-unique but do not actually represent heterogeneity between genomes. Since we are not interested in SNPs or indels in this analysis, we simply collapse any bubbles $b$ with start and end vertices $start, end$ that meet the following criteria:

\[
\begin{align*}
    b | end & \in \text{grandChildren}(start), \\
    \text{children}(start) & = \text{parents}(end), \\
    \text{orientation}(start) & = \text{orientation}(end)
\end{align*}
\]  

Note that we make sure to check orientation of $start$ and $end$ so that a bubble caused by an inversion is not lost. The PG can then be used for identifying different types of polymorphisms such as translocations (Figure 8.2).

Figure 8.1: An inversion within three synthetic genomes visualized using Mauve [23] and aligned from the PolyGraph. Note the magenta region in genome 3 that appears in the bottom half of the area designated for genome 3 in the visualizer signifying it is on the negative strand while the magenta regions in genomes 1 and 2 are on the top halves indicating the positive strand.
8.3.2 Inversion Detection

Inversions are detected easily by iterating over all vertices in the PG and examining the stored orientation for each genome at a vertex. If all genomes present at a vertex in the graph do not have the same orientation then an inversion is present. Evolutionary splits are identified by segregating the genomes present at a vertex with heterogeneous sequence orientations into forward and reverse groupings. A visualization of an inversion is shown in Figure 8.1.

8.3.3 SANS Modifications

The original SANS algorithm can be found in Wittler [126]. We modify the algorithm as follows. Because we use the PG instead of a CdBG, we encounter sets of genomes that have no supporting vertices. To account for this, we differ from the original SANS algorithm by starting each count at 1 so that evolutionary split weights are not set to 0 when computing the geometric mean (Algorithm 7).

We also modify the algorithm for use on a subset of vertices that contain inversions. These modifications include filtering vertices to check if there are inversions present and adding additional weighting to subsets of genomes present at a vertex that share the same orientation. Modifications can be seen in Algorithm 8.
Algorithm 7 SANS algorithm modified for evolutionary splits from the PolyGraph vertices. Differences include weights initializing to 1 instead of 0 (Line 1).

**Input:** Set of genomes $G$

**Output:** Weighted splits over $G$

1. $T :=$ empty map // initialize $T[S] := (1, 1)$ on first access by $S$
2. $PG :=$ PolyGraph of $G$
3. for each $v$ in $PG$ do
4. \[ S := $\text{color list of } v, \text{ where } S \subseteq G \]
5. \[ \text{if } |S| < |G|/2 \text{ then} \]
6. \[ \text{increase first element of } T[S] \text{ by length of } v \]
7. \[ \text{else} \]
8. \[ \text{increase second element of } T[G \setminus S] \text{ by length of } v \]
9. for each entry $S$ in $T$ with values $(w, w')$ do
10. output unordered split $\{S, G \setminus S\}$ of weight $\sqrt{w \times w'}$

Algorithm 8 SANS algorithm modified for inversion evolutionary splits from the PolyGraph. Differences include weights initializing to 1 instead of 0 (Line 1) and additional weighting for subsets of genomes with the same orientation in a vertex (Lines 4-13).

**Input:** Set of genomes $G$

**Output:** Weighted splits over $G$

1. $T :=$ empty map // initialize $T[S] := (1, 1)$ on first access by $S$
2. $PG :=$ PolyGraph of $G$
3. for each $v$ in $PG$ do
4. \[ S_{\text{pos}} := $\text{color list of } v \text{ with positive orientation} \]
5. \[ \text{if } |S_{\text{pos}}| < |G|/2 \text{ then} \]
6. \[ \text{increase first element of } T[S_{\text{pos}}] \text{ by length of } v \]
7. \[ \text{else} \]
8. \[ \text{increase second element of } T[G \setminus S_{\text{pos}}] \text{ by length of } v \]
9. \[ S_{\text{neg}} := $\text{color list of } v \text{ with negative orientation} \]
10. \[ \text{if } |S_{\text{neg}}| < |G|/2 \text{ then} \]
11. \[ \text{increase first element of } T[S_{\text{neg}}] \text{ by length of } v \]
12. \[ \text{else} \]
13. \[ \text{increase second element of } T[G \setminus S_{\text{neg}}] \text{ by length of } v \]
14. for each entry $S$ in $T$ with values $(w, w')$ do
15. output unordered split $\{S, G \setminus S\}$ of weight $\sqrt{w \times w'}$
8.3.4 Phylogenetic Network and Tree Inference

For generating phylogenetic networks we use SplitsTree4 [66] using the greedy weakly compatible filtering. To generate a bifurcating tree we use the scripts included by Wittler [126] which greedily filters out any conflicting ESs.

8.4 Results

For each of the datasets we generate a phylogenetic network using greedy weakly compatible filtering [65], a bifurcating phylogenetic tree using greedy compatible filtering [126] and compare the phylogenetic trees using Robinson-Foulds ($RF$) and normalized Robins-Foulds ($nRF$) metrics which measure the dissimilarity between tree topologies using ete3 [62]. All runs were completed on a machine with an Intel Xeon E5-2650v4@2.20GHz with 256GB RAM.

8.4.1 Prasinovirus

The Prasinovirus dataset is challenging due to the viruses’ high mutation rates resulting in highly diverse DNA sequence. 13 Prasinovirus genomes (Table 8.1) and the canonical tree were gathered from Finke et al. [35] using scripts provided by Wittler [126].

Because of the heterogeneity in the genomes, we use a smaller $k=17$ when generating the PolyGraph. When comparing to CdBG, we were unable to generate a useable graph with $k=17$ and resorted to $k=11$ as done by Wittler [126]. PG construction takes 0.6GB RAM. Total run-time for PG construction and ES extraction was 3 minutes. The phylogenetic networks generated from the vertex-centric and inversion only ESs can be seen in Figure 8.3. Bifurcating tree comparison with the reference tree can be seen in Figure 8.4, comparing the two we find that $RF = 8$ and $nRF = 0.4 = (8/20)$. 
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC_009899.1</td>
<td>AR158</td>
</tr>
<tr>
<td>NC_014765.1</td>
<td>BpV1</td>
</tr>
<tr>
<td>HM004430.1</td>
<td>BpV2</td>
</tr>
<tr>
<td>NC_020864.1</td>
<td>MpV-12T</td>
</tr>
<tr>
<td>NC_014767.1</td>
<td>MpV1</td>
</tr>
<tr>
<td>HQ633072.1</td>
<td>MpV-PL1</td>
</tr>
<tr>
<td>JF974320.1</td>
<td>MpV-SP1</td>
</tr>
<tr>
<td>NC_014766.1</td>
<td>OLV1</td>
</tr>
<tr>
<td>NC_013288.1</td>
<td>OtV1</td>
</tr>
<tr>
<td>NC_014789.1</td>
<td>OtV2</td>
</tr>
<tr>
<td>NC_010191.2</td>
<td>OtV5</td>
</tr>
<tr>
<td>JN225873.1</td>
<td>OtV6</td>
</tr>
<tr>
<td>NC_000852.5</td>
<td>PBCV1</td>
</tr>
</tbody>
</table>

Table 8.1: 13 genomes from the Prasinovirus dataset

![Diagram](image)

(a) all vertices  
(b) inversions

Figure 8.3: Visualization of Prasinovirus greedily extracted evolutionary splits from the PolyGraph using SplitsTree4 [66], k=17
8.4.2 E. coli

We used 27 genomes in this dataset: 20 *E. coli* genomes and 7 *Shigella* genomes (Table 8.2) and were gathered from Skippington and Ragan [118]. We constructed the PolyGraph using k=31 and took about 5.6GB RAM. Total run-time for PG construction and ES extraction was 1 hour 40 minutes. The inferred phylogenetic network can be seen in Figure 8.6. Comparison of the reference and vertex-centric inferred tree topologies can be seen in Figure 8.5 and are colored by recognized *E. coli* phylogenetic groups, we calculate $RF = 12$ and $nRF = 0.25 = (12/48)$.
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC_00091.1.fa</td>
<td><em>Escherichia coli</em> K-12 W3110</td>
</tr>
<tr>
<td>NC_000913.3.fa</td>
<td><em>Escherichia coli</em> K-12 MG1655</td>
</tr>
<tr>
<td>NC_002655.2.fa</td>
<td><em>Escherichia coli</em> O157 H7 EDL933</td>
</tr>
<tr>
<td>NC_002695.2.fa</td>
<td><em>Escherichia coli</em> O157 H7 Sakai DNA</td>
</tr>
<tr>
<td>NC_004337.2.fa</td>
<td><em>Shigella flexneri</em> 2a 301</td>
</tr>
<tr>
<td>NC_004431.1.fa</td>
<td><em>Escherichia coli</em> CFT073</td>
</tr>
<tr>
<td>NC_004741.1.fa</td>
<td><em>Shigella flexneri</em> 2a 2457T</td>
</tr>
<tr>
<td>NC_007384.1.fa</td>
<td><em>Shigella sonnei</em> Ss046</td>
</tr>
<tr>
<td>NC_007606.1.fa</td>
<td><em>Shigella dysenteriae</em> Sd197</td>
</tr>
<tr>
<td>NC_007613.1.fa</td>
<td><em>Shigella boydii</em> Sb227</td>
</tr>
<tr>
<td>NC_007946.1.fa</td>
<td><em>Escherichia coli</em> UTI89</td>
</tr>
<tr>
<td>NC_008253.1.fa</td>
<td><em>Escherichia coli</em> 536</td>
</tr>
<tr>
<td>NC_008258.1.fa</td>
<td><em>Shigella flexneri</em> 5 8401</td>
</tr>
<tr>
<td>NC_008563.1.fa</td>
<td><em>Escherichia coli</em> APEC O1</td>
</tr>
<tr>
<td>NC_009800.1.fa</td>
<td><em>Escherichia coli</em> HS</td>
</tr>
<tr>
<td>NC_009801.1.fa</td>
<td><em>Escherichia coli</em> E24377A</td>
</tr>
<tr>
<td>NC_010468.1.fa</td>
<td><em>Escherichia coli</em> ATCC 8739</td>
</tr>
<tr>
<td>NC_010498.1.fa</td>
<td><em>Escherichia coli</em> SMS-3-5</td>
</tr>
<tr>
<td>NC_010658.1.fa</td>
<td><em>Shigella boydii</em> CDC 3083-94</td>
</tr>
<tr>
<td>NC_011415.1.fa</td>
<td><em>Escherichia coli</em> SE11</td>
</tr>
<tr>
<td>NC_011601.1.fa</td>
<td><em>Escherichia coli</em> 0127 H6 E2348/69</td>
</tr>
<tr>
<td>NC_011741.1.fa</td>
<td><em>Escherichia coli</em> IAI1</td>
</tr>
<tr>
<td>NC_011742.1.fa</td>
<td><em>Escherichia coli</em> S88</td>
</tr>
<tr>
<td>NC_011745.1.fa</td>
<td><em>Escherichia coli</em> ED1a</td>
</tr>
<tr>
<td>NC_011748.1.fa</td>
<td><em>Escherichia coli</em> 55989</td>
</tr>
<tr>
<td>NC_011750.1.fa</td>
<td><em>Escherichia coli</em> IAI39</td>
</tr>
<tr>
<td>NC_011751.1.fa</td>
<td><em>Escherichia coli</em> UMN026</td>
</tr>
</tbody>
</table>

Table 8.2: 27 genomes from the *E. coli* dataset.
Figure 8.5: Trees for the *E. coli* dataset colored by recognized *E. coli* phylogenetic groups

Figure 8.6: Visualization of *E. coli* greedily extracted phylogenetic network using evolutionary splits from the PolyGraph using SplitsTree4 [66], k=31
8.4.3 Drosophila

12 *Drosophila* genomes were used (Table 8.3) gathered from Flybase [121]. We constructed the PolyGraph using k=31 using about 250GB RAM. Total run-time for PG construction and ES extraction was 26 hours and 38 minutes. The inferred phylogenetic network can be seen in Figure 8.9.

We are able to regenerate all relationships when comparing the inferred phylogenetic tree to the reference tree (Figure 8.7). Thus, \( RF = 0 \) and \( nRF = 0 = (0/18) \).

We analyzed the vertex-centric ESs from the polygraph and compared them to the ESs from a CdBG by checking concordance with a canonical *Drosophila* phylogenetic tree (Figure 8.7a). In Figure 8.8 there are two plots present: Figure 8.8a shows concordance of all ESs and Figure 8.8b shows ESs with at least 2 taxa in the split. In both cases, we see that the top 50 ESs from the PG are much more heavily weighted than those from the CdBG. We found that the most heavily weighted splits from both the PG and the CdBG often only contain a single taxon and ESs with only a single taxon are inherently correct and don’t necessarily indicate the trustworthiness of the ESs that have been generated. Thus, we provide Figure 8.8b and contrast it with Figure 8.8a to show that few of the ESs from either source have concordance with the canonical tree. Despite this, using greedy weakly compatible filtering produces trees that do agree with the canonical tree which may indicate that more heavily weighted ESs are trustworthy for inferring phylogenetic networks. The highly weighted ESs that conflict with higher weighted ESs may indicate that speciation was not the clean bifurcating event that a tree such as Figure 8.7a depicts. We believe that these ESs may be a starting point to identify reticulation events.
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Short Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>dana-all-chromosome-r1.06.fasta</td>
<td>Drosophila ananassae</td>
<td>ana</td>
</tr>
<tr>
<td>dere-all-chromosome-r1.05.fasta</td>
<td>Drosophila erect</td>
<td>ere</td>
</tr>
<tr>
<td>dgri-all-chromosome-r1.05.fasta</td>
<td>Drosophila grimshawi</td>
<td>gri</td>
</tr>
<tr>
<td>dmel-all-chromosome-r6.26.fasta</td>
<td>Drosophila melanogaster</td>
<td>mel</td>
</tr>
<tr>
<td>dmoj-all-chromosome-r1.04.fasta</td>
<td>Drosophila majavensis</td>
<td>moj</td>
</tr>
<tr>
<td>dper-all-chromosome-r1.3.fasta</td>
<td>Drosophila persimilis</td>
<td>per</td>
</tr>
<tr>
<td>dpse-all-chromosome-r3.04.fasta</td>
<td>Drosophila pseudoobscura</td>
<td>pse</td>
</tr>
<tr>
<td>dsec-all-chromosome-r1.3.fasta</td>
<td>Drosophila sechellia</td>
<td>sec</td>
</tr>
<tr>
<td>dsim-all-chromosome-r2.02.fasta</td>
<td>Drosophila simulans</td>
<td>sim</td>
</tr>
<tr>
<td>dvir-all-chromosome-r1.07.fasta</td>
<td>Drosophila virilis</td>
<td>vir</td>
</tr>
<tr>
<td>dwil-all-chromosome-r1.05.fasta</td>
<td>Drosophila willistoni</td>
<td>wil</td>
</tr>
<tr>
<td>dyak-all-chromosome-r1.05.fasta</td>
<td>Drosophila yakuba</td>
<td>yak</td>
</tr>
</tbody>
</table>

Table 8.3: 12 genomes used in the *Drosophila* dataset

(a) reference [126]  (b) compatible tree from evolutionary splits from the PolyGraph

Figure 8.7: Trees for *Drosophila*
Examining Discordant Evolutionary Splits

The PolyGraph data structure can be used beyond generating a phylogenetic tree or network by examining that vertices that contribute to a particular evolutionary splits weight. We direct our
investigation by looking at the top weighted ESs that are discordant with more heavily weighted ESs (Table 8.4). In the Drosophila dataset, we examine the top two ESs that disagree with the higher weighted ESs: one that contains *D. grimshawi* and *D. mojavensis* and another that contains *D. grimshawi* and *D. virilis* where the ES that agrees with the reference tree contains *D. mojavensis* and *D. virilis*. We further examine these ESs by re-aligning these three genomes using the PolyGraph with $k = 131$ and visualize regions of the genomes that anchor together (Figure 8.10). Examining regions that are anchored together may help indicate genomic regions that are under similar selective pressures or indicate regions of hybridization that arose as the species continued to intermix as they speciated from one another.

### 8.5 Conclusion and Future Work

In this work we have demonstrated the generation of evolutionary splits from the PolyGraph. Though there is a computational cost to identifying evolutionary splits through the PolyGraph compared to other methods (*e.g.* the colored de Bruijn Graph) the benefits of identifying genomic features through preserved genomic coordinates is worth the cost. We also demonstrate how not only can this data structure be mined to generate phylogenetic trees and networks but can also be used to further understand relationships within the tree and networks. In our future work, we’d like to move beyond the identification of inversions to other structural variants by formalizing graph structures that represent these types of polymorphisms and creating a model of evolution that would allow information derived from the PolyGraph to be used in Maximum Likelihood, Parsimony and Bayesian tree inference methods. Furthermore, evolutionary splits that can be traced back to genomic locations provide additional benefits such as discovering reticulate evolution by examining ESs that would normally be filtered out when generating a weakly compatible phylogenetic network and seeing where in genomes reticulation events may have occurred.

**Acknowledgements**

The authors would like to thank the members of the Computational Sciences Laboratory at BYU.
Figure 8.10: Regions of *D. grimshawi*, *D. mojavensis* and *D. virilis* that anchor together in the PolyGraph. An evolutionary split containing *D. grimshawi* and *D. mojavensis* and another containing *D. grimshawi* and *D. virilis* were the highest weighted discordant evolutionary splits (see Table 8.4).
<table>
<thead>
<tr>
<th>Agreement</th>
<th>ES weight</th>
<th>Species Present in ES</th>
</tr>
</thead>
<tbody>
<tr>
<td>concordant</td>
<td>5160912</td>
<td>wil</td>
</tr>
<tr>
<td>concordant</td>
<td>3441268</td>
<td>gri</td>
</tr>
<tr>
<td>concordant</td>
<td>3260119</td>
<td>ana</td>
</tr>
<tr>
<td>concordant</td>
<td>3141411</td>
<td>moj</td>
</tr>
<tr>
<td>concordant</td>
<td>2570273</td>
<td>vir</td>
</tr>
<tr>
<td>concordant</td>
<td>1956162</td>
<td>per, pse</td>
</tr>
<tr>
<td>concordant</td>
<td>1846092</td>
<td>per</td>
</tr>
<tr>
<td>concordant</td>
<td>1611894</td>
<td>sec</td>
</tr>
<tr>
<td>concordant</td>
<td>1506760</td>
<td>yak</td>
</tr>
<tr>
<td>concordant</td>
<td>1344692</td>
<td>ere</td>
</tr>
<tr>
<td>concordant</td>
<td>1087254</td>
<td>mel</td>
</tr>
<tr>
<td>concordant</td>
<td>1028323</td>
<td>gri, moj, vir</td>
</tr>
<tr>
<td>concordant</td>
<td>981559</td>
<td>ere, sim, mel, sec, yak</td>
</tr>
<tr>
<td>concordant</td>
<td>740014</td>
<td>pse</td>
</tr>
<tr>
<td>concordant</td>
<td>647494</td>
<td>gri, moj, vir, wil</td>
</tr>
<tr>
<td>concordant</td>
<td>603119</td>
<td>sim</td>
</tr>
<tr>
<td>concordant</td>
<td>542960</td>
<td>moj, vir</td>
</tr>
<tr>
<td>concordant</td>
<td>533737</td>
<td>ana, ere, sim, mel, sec, yak</td>
</tr>
<tr>
<td>concordant</td>
<td>384317</td>
<td>sec, sim, mel</td>
</tr>
<tr>
<td>concordant</td>
<td>362646</td>
<td>ere, yak</td>
</tr>
<tr>
<td>discordant</td>
<td>361097</td>
<td>gri, vir</td>
</tr>
<tr>
<td>discordant</td>
<td>346376</td>
<td>gri, moj</td>
</tr>
<tr>
<td>discordant</td>
<td>315070</td>
<td>sec, sim</td>
</tr>
<tr>
<td>discordant</td>
<td>239997</td>
<td>gri, wil</td>
</tr>
<tr>
<td>discordant</td>
<td>235899</td>
<td>ana, pse, per</td>
</tr>
<tr>
<td>discordant</td>
<td>231759</td>
<td>per, pse, wil</td>
</tr>
<tr>
<td>discordant</td>
<td>210254</td>
<td>moj, wil</td>
</tr>
<tr>
<td>discordant</td>
<td>209033</td>
<td>ana, moj, vir, wil, gri</td>
</tr>
<tr>
<td>discordant</td>
<td>197009</td>
<td>moj, vir, gri, per, pse</td>
</tr>
<tr>
<td>discordant</td>
<td>165138</td>
<td>vir, wil</td>
</tr>
</tbody>
</table>

Table 8.4: The top 30 evolutionary splits by weight calculated from the PolyGraph using the modified SANS algorithm. Most of the top weights evolutionary splits contain a single species.
Funding

This work has been supported by the Utah NASA Space Grant Consortium and EPSCoR and the BYU Graduate Research Fellowship.
References


[81] Heng Li. wgsim-read simulator for next generation sequencing, 2011.


[104] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher,


