COMPUTATIONAL PAN-GENOMICS: ALGORITHMS AND APPLICATIONS

by

Alan Michael Cleary

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2018

©COPYRIGHT

by

Alan Michael Cleary

2018

All Rights Reserved

DEDICATION

I dedicate this dissertation to my parents, Mike and Joyce, who have always encouraged me in all my endeavors. And to my sister, Beth, who has always reminded me to stay true to my passions. Thank you.

ACKNOWLEDGEMENTS

I would like to thank my Advisor, Dr. Brendan Mumey, for his support and guidance. I would also like to thank Andrew Farmer at the National Center for Genome Resources for his support and patience with my feeble understanding of biology. Lastly, I would like to thank my undergraduate advisor, Dr. John Peterson, for taking me under his wing and giving me the opportunities that prepared me to pursue a graduate degree.

Funding Acknowledgment

This work was kindly supported by the National Center for Genome Resources, United States Department of Agriculture Agricultural Research Service project funding for the Legume Information System, National Science Foundation Integrative Organismal Systems award number 1444806, National Science Foundation Advances in Biological Informatics award number 1542262, Google Summer of Code, and by a Ph.D Dissertation Completion Award provided by the Montana State University Graduate School. Thank you.

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1 Preliminaries 2
1.2 Pan-Genomics 6
1.3 Research Questions 7
1.4 Contributions 8
1.5 Organization 9

2. APPROXIMATE FREQUENT SUBPATHS 10

2.1 Contributions 10
2.2 Introduction 10
2.3 Related Work 11
2.4 Problem Definition 12
2.4.1 Problem Complexity 16
2.5 Algorithm 18
2.5.1 Time complexity 19
2.6 Experimental Results 20
2.6.1 *Saccharomyces cerevisiae* 20
2.7 Conclusions 25

3. FREQUENTED REGIONS 27

3.1 Contributions 27
3.2 Introduction 28
3.3 Related Work 28
3.4 Problem Definition 30
3.4.1 Problem Complexity 33
3.5 Algorithm 34
3.5.1 Reverse-Complement Support 36
3.5.2 Weighted Support 38
3.5.3 Parallelization 39
3.5.4 Time Complexity 42
3.6 Applications 43
3.6.1 Machine Learning with FRs 43
3.6.2 Graph Simplification 45
3.7 Experimental Results 46
3.7.1 *Staphylococcus aureus* 47
3.7.2 *Saccharomyces cerevisiae* 48
3.7.2.1 Parallelization Speedup 49

TABLE OF CONTENTS – CONTINUED

3.7.2.2 Consistency with Yeast Biology ... 52
3.7.2.3 Using FRs for Classification ... 58
3.7.2.4 Visualizing Pan-Genomic Space ... 60
3.8 Conclusions ... 62

4. GENOME CONTEXT VIEWER ... 64

4.1 Contributions ... 64
4.2 Introduction ... 65
4.3 GCV Application ... 66
4.4 Related Work ... 72
4.5 Architecture ... 76
4.5.1 Technology Stack ... 76
4.5.2 Services ... 78
4.6 Algorithms ... 79
4.6.1 Track Search via AFS Supporting Paths ... 79
4.6.1.1 Time Complexity ... 81
4.6.2 Merging Alignments ... 81
4.6.2.1 Time Complexity ... 82
4.6.3 Alignment Coordinates ... 83
4.6.3.1 Time Complexity ... 83
4.6.4 Clustering Tracks ... 84
4.6.4.1 Time Complexity ... 85
4.6.5 Multiple Alignment of Tracks ... 86
4.6.5.1 Time Complexity ... 90
4.6.6 Dynamic Chromosome-Scale Synteny Blocks ... 90
4.6.6.1 Time Complexity ... 93
4.6.7 Track Packing ... 93
4.6.7.1 Time Complexity ... 94
4.7 Experimental Results ... 94
4.7.1 Block Resolution ... 95
4.7.2 LIS and LegFed Integration ... 96
4.8 Conclusions ... 98

5. CONCLUSION AND FUTURE WORK ... 99

5.1 Future Work ... 99
5.1.1 Identifying Interesting Biology ... 99
5.1.2 Furthering Frequented Regions ... 101
5.1.3 Extending the Genome Context Viewer ... 103

TABLE OF CONTENTS – CONTINUED

5.1.4 Pan-Genome Representation 104
5.1.5 Additional Directions 106
5.2 Closing Remarks 107

REFERENCES CITED 109

APPENDIX: SCHOLARLY PRODUCTS 119

LIST OF TABLES

Table Page

1.1 DNA length measurements and abbreviations. 3

2.1 10 yeast strains. 21

3.1 `Sibelia` versus `FindFRs` on 4 strains of *Staphylococcus aureus*. 48

3.2 48 yeast strains. 50

3.3 The size of each yeast CDBG and the number of iFRs found. 51

LIST OF FIGURES

Figure Page

1.1 The cost of sequencing a human-sized genome per year.......................2

1.2 Deoxyribonucleic acid.......................4

1.3 The construction of a pan-genomic de Bruijn graph.......................5

2.1 Example of an AFS.......................15

2.2 Multiple sequence alignment of HFI1 genes in yeast.......................23

2.3 A phylogenetic tree of HFI1 genes in yeast.......................24

3.1 Example FRs.......................33

3.2 Distribution of yeast iFR support versus average length.......................51

3.3 Running times of the linear and parallel maximal graph matching algorithms.......................52

3.4 FR representation of three novel yeast insertions.......................54

3.5 The 17kb insertions found in EC1118 (yeast).......................56

3.6 Box plot of the AUROC curve results for SVMs classifying yeast using their FR content as features.......................60

3.7 Box plot of the AUROC curve results for an SVM classifying yeast using their (un)filtered FR content as features.......................61

3.8 Yeast industrial usage multidimensional-scaling plot.......................62

4.1 A GCV search view.......................69

4.2 A GCV multi view multiple sequence alignment.......................70

4.3 A GCV multi view Circos style plot.......................71

4.4 Comparison of an inverted region on soybean chromosomes 1 and 2 produced using the PGDD locus search.......................74

4.5 The software architecture and data flow of the GCV.......................77

4.6 The topology of the profile HMM used for multiple sequence alignment of GCV tracks.......................87

LIST OF FIGURES – CONTINUED

Figure Page

4.7 Comparison of results from MCScanX using gene family assignments to those produced using DAGchainer on CDS pairwise matches. 97

LIST OF ALGORITHMS

Algorithm Page

3.1 Compute FR supporting path segments. 35

3.2 Evaluate FR merge. 36

3.3 Agglomerate FRs. 37

3.4 Report interesting FRs. 38

3.5 Linear maximal weighted graph matching. 40

3.6 Parallel maximal weighted graph matching. 41

ABSTRACT

As the cost of sequencing DNA continues to drop, the number of sequenced genomes rapidly grows. In the recent past, the cost dropped so low that it is no longer prohibitively expensive to sequence multiple genomes for the same species. This has led to a shift from the single reference genome per species paradigm to the more comprehensive pan-genomics approach, where populations of genomes from one or more species are analyzed together.

The total genomic content of a population is vast, requiring algorithms for analysis that are more sophisticated and scalable than existing methods. In this dissertation, we explore new algorithms and their applications to pan-genome analysis, both at the nucleotide and genic resolutions. Specifically, we present the Approximate Frequent Subpaths and Frequented Regions problems as a means of mining syntenic blocks from pan-genomic de Bruijn graphs and provide efficient algorithms for mining these structures. We then explore a variety of analyses that mining synteny blocks from pan-genomic data enables, including meaningful visualization, genome classification, and multidimensional-scaling. We also present a novel interactive data mining tool for pan-genome analysis — the Genome Context Viewer — which allows users to explore pan-genomic data distributed across a heterogeneous set of data providers by using gene family annotations as a unit of search and comparison. Using this approach, the tool is able to perform traditionally cumbersome analyses on-demand in a federated manner.

CHAPTER ONE

INTRODUCTION

Since the publication of the first working draft of the human genome in 2001 [16], the National Human Genome Research Institute (NHGRI), a division of the National Institutes of Health (NIH), has been tracking the cost of sequencing a human-sized genome each year, as shown in Figure 1.1. Not only has the cost of sequencing a human genome significantly decreased, starting in January of 2008, the rate at which the cost was decreasing surpassed Moore's Law [76]. Subsequently, the number of genomes sequenced per year has been growing at an exponential rate [86], with research institutions and genome sequencing consortia sequencing multiple genomes per species at a massive scale.[1]

Although the cost of sequencing genomes has drastically declined in the past few years, the common approach to genomics is still "reference-centric"; that is, a single *reference* genome is used as a representative for a particular species. Since a reference genome is the sequence of a single individual or a mosaic of individuals as a single linear sequence, this approach is biased. It is a relic of the foundational period of genomics when sequencing multiple genomes of a particular species was limited by technological and budgetary concerns. Only recently has genomics begun to expand from a single reference per species paradigm into a more comprehensive pan-genome approach that analyzes multiple individuals together.

As a young field, there is still much foundational work to be done [76].

[1] See `http://www.1000genomes.org/` `http://www.1001genomes.org/`, `http://www.100kgenome.vetmed.ucdavis.edu/`, and [50, 85].

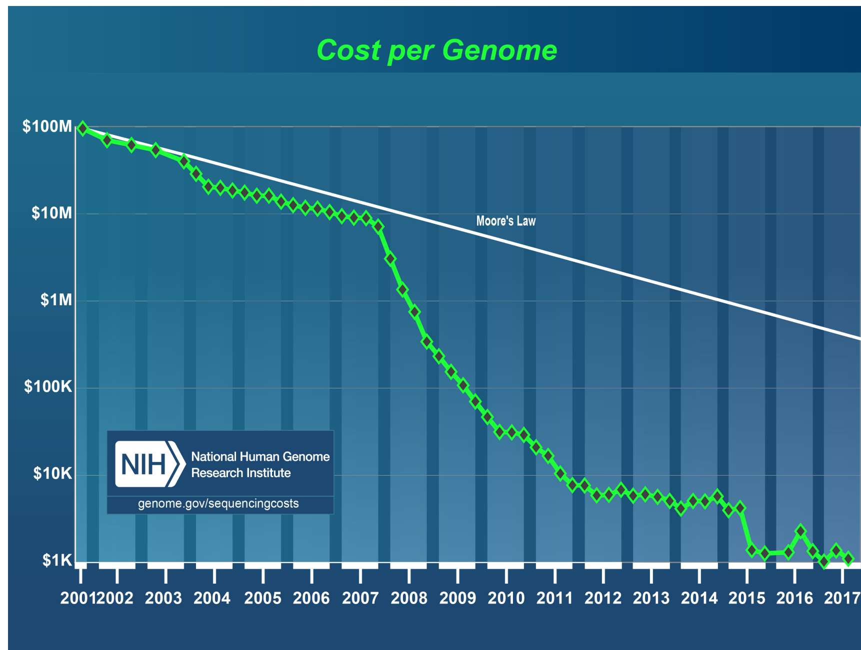

Figure 1.1: The cost of sequencing a human-sized genome per year according to the NHGRI [43].

Specifically, there is a need for efficient data structures, algorithms, and statistical methods to perform bioinformatic analyses of pan-genomic data. Addressing these computational needs is the focus of this dissertation.

1.1 Preliminaries

Deoxyribonucleic acid (DNA) is the molecule that encodes the "blueprint" of all known living organisms and many viruses – see Figure 1.2 for an artistic depiction. A sequence of DNA is composed of monomer units called nucleotides. Each nucleotide is composed of one of four nucleobases: cytosine (C), guamine (G), adenine (A), and thymine (T). Most DNA molecules consist of two strands coiled around each other to form a double helix. For each nucleotide in one strand, there is a complementary nucleotide in the other strand: A $\rightarrow$ T, C $\rightarrow$ G, G $\rightarrow$ C, and T $\rightarrow$ A. A nucleotide and

Table 1.1: DNA length measurements and abbreviations.

Unit	Abbreviation	Length
base pair(s)	bp	A single DNA nucleotide and its compliment
kilo base pairs	kb	$1,000$bp
mega base pairs	Mb	$1,000,000$bp
giga base pairs	Gb	$1,000,000,000$bp

its complement are referred to as a *base pair*. The units of length that are typically used to describe the length of a sequence of DNA are listed in Table 1.1.

In this work, we discuss de Bruijn graphs of the variety commonly used for genome assembly, rather than the classical data structure from which they are derived [21, 31]. Here, a de Bruijn graph is an abstract graph that represents overlap information in a set of DNA sequences [41, 88]. An example is shown in Figure 1.3. In a de Bruijn graph, each unique length k subsequence (k-mer) in the set of DNA sequences (Figure 1.3A) is represented by a node in the de Bruijn graph (Figure 1.3B). If the last $k-1$ characters of a k-mer overlap with the first $k-1$ characters of another k-mer in one or more of the DNA sequences in the set, then the corresponding de Bruijn nodes are connected by a directed edge, the orientation of which reflects the k-mers' ordering in the DNA sequences. A de Bruijn graph can be compressed by collapsing non-divergent chains of nodes into a single super-node [51] (Figure 1.3C). Additionally, a de Bruijn graph can be colored by giving each genome's path through the graph a unique color [44]. A bubble in a de Bruijn graph is a pair of subpaths that have the same start and end nodes but are otherwise vertex disjoint (Figure 1.3B and C).

[2] `https://www2.le.ac.uk/projects/vgec/highereducation/topics/dna-genes-chromosomes`

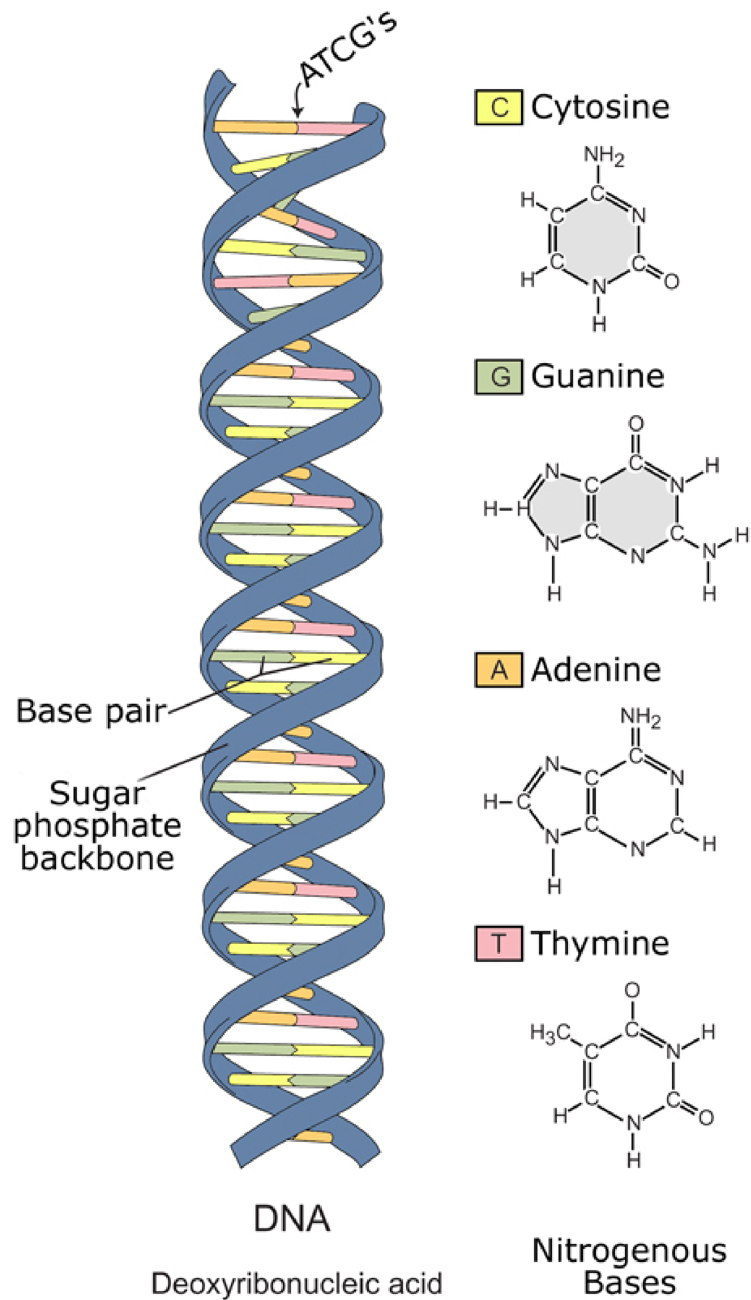

Figure 1.2: Deoxyribonucleic acid (DNA). (left) A double helix composed of two strands of nucleotides. (right) The four nucleobases that nucleotides are composed of: cytosine (C), guamine (G), adenine (A), and thymine (T). Image from the Virtual Genetics Education Centre, University of Leicester.[2]

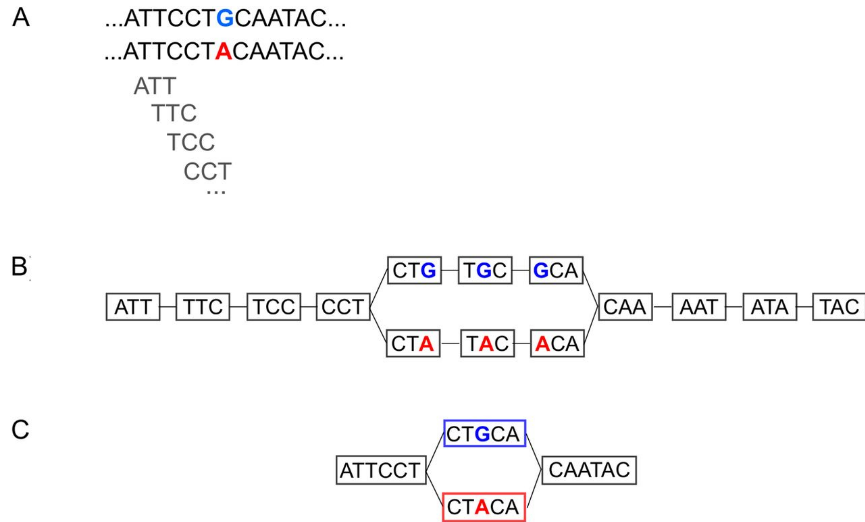

Figure 1.3: The construction of a pan-genomic de Bruijn graph. (A) Two sequence segments illustrating a SNP (red in first sequence and blue in second sequence) and their length 3 k-mers. (B) A de Bruijn graph constructed from the k-mers of the sequence segments illustrating a bubble formed by the SNP. (C) The same de Bruijn graph with its non-divergent chains of nodes collapsed into super-nodes. Image adapted from [67].

A *repeat* is an exact sequence of nucleotides or genes that occurs multiple times in a genome. Similarly, a *synteny block* is a sequence of nucleotides or genes that is present, or *conserved*, across related genomes. Unlike repeats, instances of a synteny block may vary within or between genomes. Identifying synteny blocks and their variations is a fundamental bioinformatics analysis, for example, they are integral to inferring phylogenies and characterizing functional variation within a group of related species [26].

Variation in a genome refers to deviation from another sequence, typically a reference. The basic types of variation at the nucleotide resolution are single nucleotide polymorphisms (SNPs, pronounced “snips”), the insertion or deletion of contiguous sequence (indels), novel repeats, and inversions – the reversal of a repeat. In a de Bruijn graph, such variations often correspond to bubbles, such as the bubble

formed by the SNP in Figure 1.3. When such polymorphisms are observed in the sequence of a gene, then the genes are referred to as *homologous* – having evolved from the same common ancestor. More specifically, if the genes are in different species but evolved from the same common ancestor, then they are *orthologous*. Alternatively, if the genes diverged within the same species, then they are *paralogous*.

Variations at the genic resolution are called structural variations and include the presence, absence, and variation of copy number,[3] polyploidy,[4] segmental and/or tandem duplication,[5] and inversions.

1.2 Pan-Genomics

The term "pan-genome" was first coined by Sigaux in [99] and was used to describe a public database containing an assessment of genome and transcriptome alterations in major types of tumors, tissues, and experimental models. The term was later revitalized by Tettelin et al. in [104] to describe a microbial genome by which genes were in the core (present in all strains) and which genes were dispensable (missing from one or more of the strains). The term is now used as an umbrella to describe any collection of genetic or genomic sequences from the same or similar species to be analyzed jointly or to be used as a reference.

Today, a variety of tools exist for pan-genomic analysis [106]. Unfortunately, they are limited in the size and number of genomes they can analyze. This has generally been due to insufficient computing hardware and improper representation of the data (non-graphical). Fortunately, due to recent advances in computing hardware and algorithms, these are no longer blocking issues.

[3]Copy number refers to the number of times a gene is repeated.

[4]Polyploidy refers to the presence of more than two paired sets of chromosomes in an organism.

[5]Segmental duplication refers to the near perfect replication of a sequence of genes. Tandem duplication refers to the (segmental) duplication being collocated with the duplicated sequence.

1.3 Research Questions

The questions this dissertation will address are:

Question 1

What data structures, algorithms, and statistical methods can be used to perform bioinformatic analyses of pan-genomic data?

Given the breadth of this question, we will focus on the following sub-questions:

Question 2

Pan-genomic data are commonly represented with a graph data structure; how can this data structure be utilized to enable novel and efficient methods of bioinformatic analysis?

Question 3

Given the potentially massive size of pan-genomic data, how can these data structures and algorithms be distributed/parallelized?

Question 4

The interpretation of biological data and the results of their analysis are tasks performed by domain experts, often with the aid of visualization tools. How can pan-genomic data and the results of their analysis be visualized effectively for the purpose of interpretation?

1.4 Contributions

This dissertation contributes to the area of computational biology, specifically, computational pan-genomics. It addresses the research questions in Section 1.3 with the following contributions:

1. The definition of the Approximate Frequent Subpaths problem and its application to pan-genomic de Bruijn graphs as a means of identifying synteny blocks. This includes presenting a novel algorithm for identifying Approximate Frequent Subpaths with experiments on a real data set, demonstrating the efficacy of using path-graph methods as a means of analyzing pan-genomic data.

2. The definition of the Frequented Regions problem and its application to pan-genomic de Bruijn graphs as a means of identifying synteny blocks. This includes presenting novel efficient, parallelized algorithms for identifying Frequented Regions with experiments on real data sets, further demonstrating the efficacy of path-graph methods as a means of analyzing pan-genomic data. Additionally, we present a variety of analyses that mining synteny blocks from pan-genomic data enables, including a probability density function for ranking Frequented Regions by their probabilities, meaningful visualization, genome classification with Support Vector Machines, and multidimensional-scaling using the Canberra distance.

3. A novel interactive data mining tool — the Genome Context Viewer — for pan-genome analysis. To our knowledge, this is the first tool to use gene family annotations to perform on-demand pan-genomic analyses of genomes distributed across a heterogeneous set of data providers – data federation. These analyses are performed with pairwise and multiple sequence alignment

algorithms for which we present novel extensions to enable the identification of different types of structural variation. We also present a novel profile Hidden Markov Model configuration for computing multiple sequence alignments on annotated gene sequences as well as a novel algorithm for computing pairwise synteny blocks between chromosomes represented as annotated gene sequences.

See the Appendix for the complete list of scholarly products that this dissertation is composed of.

1.5 Organization

The research questions in Section 1.3 are addressed in the following Chapters: Chapter 2 addresses Questions 2 through 4, with an emphasis on Question 2, by introducing the Approximate Frequent Subpaths problem, as outlined in Contribution 1. Chapter 3 addresses Questions 2 through 4, with an emphasis on Questions 2 and 3, by introducing the Frequented Regions problem, as outlined in Contribution 2. Chapter 4 addresses Questions 2 through 4, with an emphasis on Questions 3 and 4, by introducing the Genome Context Viewer, as outlined in Contribution 3. And in Chapter 5, we provide our conclusions and discuss future work.

CHAPTER TWO

APPROXIMATE FREQUENT SUBPATHS

Mining synteny from populations of genomes is an important problem. In this Chapter, we address the synteny mining problem, and subsequently research Questions 2 through 4 from Chapter 1, with an emphasis on Question 2, by considering the problem of mining Approximate Frequent Subpaths (AFSs) from a pan-genome de Bruijn graph, where each genomic sequence corresponds to a path in the graph.

2.1 Contributions

Expanding on Contribution 1 from Chapter 1, we formalize the AFS problem, discuss its computational complexity, and describe an effective algorithm for mining AFSs. We also discuss results of the algorithm applied to a *Saccharomyces cerevisiae* (yeast) pan-genome data set, which corroborate the existing yeast knowledge base, thus demonstrating the efficacy of using path-graph methods as a means of analyzing pan-genomic data.

2.2 Introduction

Recently, pan-genomic data have been represented using colored de Bruijn graphs [5, 71, 75], where each genome's path through the graph is given a unique color [44]. We propose a novel method of path-graph analysis based on *Approximate Frequent Subpaths* (AFSs), that is, subpaths through a graph that are approximately followed by some subset of the paths through the graph, and use it to analyze the pan-genome of 10 yeast genomes. We chose yeast as our target organism, because it

is a well-studied model system with some published comparative genomic and pan-genomic work. This allowed us to use the existing knowledge base to assess the quality of the subpaths found by our algorithm.

2.3 Related Work

The AFS problem is similar to the Frequent Itemset Mining (FI) problem, which identifies sets of items that frequently occur together in a database (matrix) of transactions [1]. Unlike the AFS problem, FI does not consider the structure of the graph or the ordering of nodes when mining itemsets. There are variations of FI that are tolerant to noise in the data [11, 64]. These require that the supporting transactions (rows) in the transaction matrix meet a *row error threshold* constraint and the items (columns) meet a *column error threshold* (support) constraint.

A graph-specific problem similar to the AFS problem is Frequent Subgraph Mining (FS), which is concerned with finding frequent subgraphs in a database of graphs. It is a well-studied problem [47] that is commonly applied to biological data sets [36, 40, 53]. Unfortunately, the existing methods are not tolerant to error, nor do they scale to pan-genomic data. For example, in [36] the data sets on which the authors evaluate their methods are equivalent in size to what would be considered a small pan-genome, such as a microbial pan-genome [71, 74]. A graph-specific problem more closely related to the AFS problem is Exact Frequent Subpath Mining (EFS) [33]. While EFS is different from FI in that it considers the structure of the graph to achieve more efficient running time and more accurate results, it is incapable of mining frequent subpaths whose supporting paths contain some error.

Recently, bioinformatics tools have started to explore multiple genomes in analyses. Several tools have evolved to take advantage of information from multiple, closely related genomes (species, strains/lines) to perform bioinformatic analyses,

such as variant detection without the bias introduced from using a single reference. For example, Cortex [44] uses colored de Bruijn graphs for *de novo* assembly[1] and genotyping of variants across samples, intelligently using information across species to improve the analysis for each individual. Specifically, it characterizes different types of bubbles (such as the one depicted in Figure 1.3 in Chapter 1) within a population, but only on a per bubble basis, whereas an AFS may contain several bubbles that compose a larger, more interesting biological structure. Another tool, Sibelia [74] uses a heuristic approach to identify nested syntenic blocks within strains of the same species. Though Sibelia does not require a reference, it cannot scale beyond small populations of microbial genomes, and the results are likely to include spurious alignments due to the repeated modification of the input sequence to facilitate the progression of the algorithm [89].

2.4 Problem Definition

Given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, a *path* is a sequence of nodes $p = \langle n_1, n_2, \ldots, n_L \rangle$ such that $\forall n_i \in p$, $n_i \in V$ and $\forall n_i, n_{i+1} \in p$, $(n_i, n_{i+1}) \in E$. An approximate frequent subpath (AFS) is characterized by a path and a set of supporting paths that generally traverse the path, meaning the path approximates a subpath in each of the supporting paths. An approximate subpath is *frequent* if there are a sufficient number of paths supporting it. We assume the following input and parameters are supplied:

- A graph G and set of paths P within G.

  In this application, G corresponds to a compressed de Bruijn graph (CDBG) for a given pan-genome (as described in Chapter 1). Each genomic sequence

[1] De novo assembly is the assembly of a genome without the aid of a reference.

corresponds to a path p in G; P is the collection of these paths.

- Parameters: $\epsilon_r, \epsilon_c, \in [0,1]$, $\mu_i \geq 0, \text{minsup} \in \mathbb{N}$

  The *row error threshold*, ϵ_r, controls the fraction of nodes on a subpath are allowed to be missing from a supporting path. Similarly, the *column error threshold*, ϵ_c, controls the fraction of supporting paths that must support any one node in the AFS. The insertion error threshold, μ_i, controls how many nodes can be "inserted" into a supporting path. A supporting path node is considered inserted if it does not belong to the AFS but lies between two nodes that do.

Suppose $p = \langle n_1, n_2, \ldots, n_L \rangle \in P$, where n_i is the ith node visited by p and L is the length of the path. We define the subpath $p[i,j] = \langle n_i, n_{i+1}, \ldots, n_j \rangle$. Two subpaths $p[i,j]$ and $p[i',j']$ are *index-disjoint* if and only if their index intervals are disjoint, that is, $[i,j] \cap [i',j'] = \emptyset$.

Definition 1. *An* approximate frequent subpath *(AFS) is a tuple* (p_a, S)*, where* p_a *is some path in* G*, referred to as an* anchor path*, and* S *is a set of supporting subpaths from paths in* P*.*

Note, p_a is just a path in G and need not be a subpath of any path in P. We define the requirements on S as as follows:

A subpath $p[i,j]$ *supports* an anchor path p_a if the following conditions are met:

$$|\text{match}(p_a, p[i,j])| \geq (1 - \epsilon_r) \cdot |p_a| \tag{2.1}$$

$$|p[i,j]| - |\text{match}(p_a, p[i,j])| \leq \mu_i \cdot |p_a|, \tag{2.2}$$

where

$$\text{match}(p_1, p_2) = p_1 \cap p_2, \tag{2.3}$$

treating p_1 and p_2 as multisets of nodes, so that repeated nodes are counted. Lastly, we require the subpaths in S to be index-disjoint and

$$|\{p[i,j] \in S : n \in \text{match}(p_a, p[i,j])\}| \geq (1 - \epsilon_c) \cdot |S|, \forall n \in p_a \tag{2.4}$$

$$|S| \geq \text{minsup}. \tag{2.5}$$

Note, by this formulation, supporting subpaths need not be simple.

Here, constraint (2.1) says that each subpath $p[i,j]$ must contain all but an ϵ_r-fraction of the anchor path's nodes, and constraint (2.2) says that the number of inserted nodes in the subpath (the LHS of (2.2)) is at most some multiple $\mu_i \geq 0$ of the length of the anchor path. Constraint (2.4) requires that each node in the anchor path is contained in all but an ϵ_c-fraction of the supporting subpaths in S and constraint (2.5) enforces that the number of supporting subpaths in S meets the minimum support requirement.

We define the *support* of an AFS as

$$\text{support}(p_a, S) = |S|. \tag{2.6}$$

Our goal is to find AFSs that have high support. As such, an important point to consider is that, unlike FIs, AFSs do not necessarily meet the Apriori Property [1], that is, subpaths of a valid AFS anchor path are not necessarily valid AFS anchor paths themselves.

Lemma 2.4.1. *Subpaths of a valid AFS anchor path are not necessarily valid AFS anchor paths themselves.*

Proof. This can be proven directly from the AFS problem definition. Observe that every supporting subpath of a valid anchor path need not support every node in the

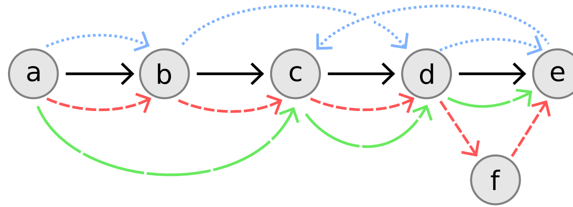

Figure 2.1: Example of an AFS: The anchor path is $p_a = \langle a, b, c, d, e \rangle$ and there are three potential supporting subpaths. If minsup $= 3$, $\epsilon_r = \frac{1}{5}$, $\epsilon_c = \frac{1}{3}$, and $\mu_i = \frac{1}{5}$, then p_a is a valid AFS. Conversely, if ϵ_r, ϵ_c, or μ_i are set to 0, then p_a is not a valid AFS. Figure created by Brittany Fasy.

anchor path, but rather an ϵ_r fraction of the nodes. This means a supporting subpath may not necessarily support an ϵ_r fraction of the nodes in some subpath of the anchor path. Therefore, each supporting subpath of a valid anchor path may not necessarily support each of the anchor path's subpaths, implying the subpaths of the anchor path are not necessarily valid AFS anchor paths themselves. □

Conversely, if $\epsilon_r = 0$ or $\epsilon_c = 0$, then AFSs meet the Apriori Property.

Lemma 2.4.2. *If $\epsilon_r = 0$ or $\epsilon_c = 0$, then subpaths of a valid AFS anchor path are themselves valid AFS anchor paths.*

Proof. This can be proven directly from the AFS problem definition. Observe, when $\epsilon_r = 0$ or $\epsilon_c = 0$, then every supporting subpath of a valid anchor path must support every node in the anchor path. This means a supporting subpath must support each node in any valid subpath of the anchor path. Therefore, each supporting subpath of a valid anchor path must support each of the anchor path's subpaths, implying the subpaths of the anchor path are themselves valid AFS anchor paths. □

Figure 2.1 shows an example AFS.

2.4.1 Problem Complexity

Counting exact Frequent Itemsets is known to be #P-complete [112]. It can be seen that the exact FI problem can be reduced to the AFS problem.

Lemma 2.4.3. *Counting the number of AFSs in a graph is #P-complete.*

Proof. The proof is reduction from the FI problem. Given an instance of the FI problem, (I, T), where I is a set of items and T is a set of transactions — each containing one or more items from I — create a connected graph G with a vertex for each item in I and a set of paths P containing a path for each transaction in T. The ordering of the vertices in each path in P does not matter. By Lemma 2.4.2, we let $\epsilon_r, \epsilon_c = 0$ and $\mu_i = |G|$ so that AFSs in (G, P) meet the Apriori Property. By this reduction, there is a one-to-one correspondence between FIs in (I, T) and AFSs in (G, P), so both problems have the same number of solutions – the reduction is parsimonious. This means any algorithm that can count the number of AFSs in (G, P) can be used to count the number of Frequent Itemsets in (I, T), and vice versa. Therefore, counting the number of AFSs in a graph is #P-complete. □

Furthermore, the problem of deciding whether there exists a set S of supporting paths for a given candidate anchor path p_a is NP-complete:

Lemma 2.4.4. *Deciding if there exists a set S of supporting paths for a given candidate anchor path p_a is NP-complete.*

Proof. Given a candidate anchor path p_a and some set of paths S, it can be determined whether or not S supports p_a in polynomial time. First, if $|S| <$ minsup, then S is not a supporting set of p_a. Next, it can be determined if the paths in S support p_a by iterating each path and identifying non-overlapping intervals that cover at least an ϵ_r fraction of the nodes in p_a and have insertions of length at most μ_i. This is

the bottleneck of the algorithm, taking $O(L|p_a|)$ time, where L is the total length of the paths in S. If one or more paths in S have no such interval, then S is not a supporting set of p_a. Once all such intervals have been identified, each node in p_a must be iterated to determine whether or not it is traversed by an ϵ_c fraction of the paths in S. If so, then S is a supporting set of p_a.

The remainder of the proof is reduction from SET-COVER. Given a SET-COVER instance, (X, C, k), where X is a set of elements and C is a collection of sets of elements, we want to know whether there is a cover $C' \subset C$ such that C' covers X and $|C'| \leq k$. Construct a graph G whose nodes are X plus an additional new node y. Let $p_a = \langle x_1, \ldots, x_n, y \rangle$. Let P be defined as follows: For each $A \in C$ construct a path p_A that visits the nodes in A (and only those nodes). Add an additional trivial path p_y that just visits node y. Let $\epsilon_r = 1, \epsilon_c = 1 - \frac{1}{k+1}, \mu_i = 0, \text{minsup} = 1$ and $\text{match}_{\text{set}}$ is the path matching function. Because of constraint (2.4), p_y must belong to any solution S. For $n = y$, the LHS of (2.4) is 1, no matter what other paths are added to S. Thus, $1 \geq \frac{1}{k+1}|S|$, or equivalently, $|S| \leq k + 1$. Finally, in order for (2.4) to hold for the other nodes of p_a, each node $x \in p_a$ must be visited by at least one path in S. Thus, there does not exist a set S of supporting paths for p_a in G unless there exists a set cover $C' \subset C$ such that $|C'| \leq k$.

Conversely, we observe that if anchor path p_a in G has a set S of supporting paths, then a set $C' \subset C$ can be constructed such that C' covers X and $|C'| \leq k$. This is because the previously defined mapping between the collection of sets C and the collection of paths $P \setminus \{p_y\}$ is a bijection, meaning for each path in $S \setminus \{p_y\}$ there is a set in C that corresponds to the same nodes in G. The collection of all such sets is C', so $|C'| = |S| - 1$. Since $\epsilon_c > 0$, each node in G exists in at least one path in S and, therefore, in at least one set in C', other than y, thus covering X. By construction, node y is only ever supported by path p_y, and path p_y always supports node y and

no others. We have shown that this constraint yields the inequality $1 \geq \frac{1}{k+1}|S|$, or equivalently, $|S| \leq k+1$. Therefore, since $|C| = |S| - 1$, it follows that $|C'| \leq k$. □

2.5 Algorithm

Our AFS-finding algorithm uses a bottom-up approach to determine candidate anchor paths of increasing length. This search is organized using a search tree T, with the root representing an empty path. The first level in T represents all paths of length 1, in other words, each node in the CDBG G is the starting point of some path. Each tree node v has branches to all neighboring nodes in G. Thus, a path from the root to v represents a path in G. We use a greedy strategy to explore T, using a priority queue Q to maintain a *frontier* of nodes whose children have not yet been explored. Nodes in Q are ordered by their support (2.6); so the most promising frontier nodes are explored first. Since the size of T is exponential, we limit the number of nodes expanded to a user-specified maximum value and do not explore a node further if its support falls below minsup. We note that by Lemma 2.4.1 even if we do not limit the number of nodes expanded, this approach would not necessarily discover all AFSs in the graph.

By Lemma 2.4.4, it is NP-hard to determine the set of supporting subpaths for a given anchor path p_a. In order to cope with this, we relax the problem to allow *fractional* supporting subpaths. The idea is to introduce a real-valued variable $x_{p[i,j]} \in [0,1]$ for each supporting subpath $p[i,j] \in S$. This formulation allows for an efficient approach to checking if a candidate anchor path forms an AFS using linear programming as outlined below:

1. Given a candidate p_a, for each path $p \in P$: compute a set of potential supporting

subpaths

$$C_p = \{p[i,j] : p[i,j] \text{ satisfies (2.1) and (2.2)}\}.$$

This can be done efficiently, for instance, by embedding each path in the graph during construction, specifically, by having each vertex store the segments of the paths that traverse them. Then, given a candidate subpath p_a, computing a set of potential supporting subpaths C_p becomes a matter of simple arithmetic. Let $C = \cup_p C_p$ be all possible candidate supporting subpaths.

2. Create a $0, 1$ matrix M that has $|p_a|$ columns. For each $p[i,j] \in C$, add a new row r to M using the following rule: if node $p_a[k] \in \text{match}(p_a, p[i,j])$, then put a 1 in column k, otherwise put a 0.

3. The problem is now to select a subset of the rows of M that corresponds to an index-disjoint collection of subpaths meeting constraints (2.4) and (2.5). The subsets $p[i,j]$ corresponding to these rows will form S. Observe that this problem can be formulated as an integer linear program (ILP) involving variables $x_{p[i,j]}$ for all paths $p[i,j] \in C$. By allowing fractional support, this ILP becomes an LP that can be solved efficiently.

We note that the problem of finding candidate anchor paths remains hard. This is because by Lemma 2.4.1 it is not always possible to find longer AFSs by extending the anchor paths of shorter AFSs by a fixed length.

2.5.1 Time complexity

As mentioned before, the size of search tree T is exponential relative to the size of the graph, so our algorithm has exponential run-time. By bounding the number of expansions that can be performed in the tree and using a relaxed ILP to compute

sets of supporting paths, the algorithm can still be used to find interesting results on real data sets, as will be shown in Section 2.6.

2.6 Experimental Results

We implemented our algorithm as a Java program, using CPLEX[2] to solve the linear programs for computing fractional support and taking advantage of obvious opportunities for parallelization. Specifically, the exploration of different subpaths in the search tree T and the computation of supporting subpaths are independent of other such computations, enabling these tasks to be performed in parallel. All experiments were performed on an HP DL580 Symmetric multiprocessing (SMP) 64-core server with 1 TB RAM. The algorithm was able to identify interesting AFSs for the yeast data set in a few hours.

2.6.1 *Saccharomyces cerevisiae*

Saccharomyces cerevisiae (yeast) is a well-studied model system with some published pan-genomic work [8, 24]. With a genome size of approximately 12Mb, it is tractable for algorithm testing. It is highly diverse, economically relevant, and its multiple industrial applications enable interesting functional genomics. Our yeast pan-genome was built with assemblies from the *Saccharomyces* Genome Database.[3] The original data set consisted of 55 assemblies from 48 yeast strains. For testing purposes, we created a subset from a wide geographical range. This subset consists of 10 strains, including 6 strains used in wine-making and 4 strains used in bread-making, as shown in Table 2.1.

We searched for highly conserved AFSs, that is, sequences that are highly similar,

[2] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[3] http://www.yeastgenome.org/

Table 2.1: The 10 yeast strains with usage and region listed where known.

Strain	Source	Region
AWRI796	Wine	South Africa
BC187	Wine	United States
CLIB215	Bakery	New Zealand
CLIB324	Bakery	Vietnam
DBVPG6044	Wine	West Africa
L1528	Wine	Chile
LalvinQA23	Wine	Portugal
Red Star	Bakery	United States
VL3	Wine	France
YS9	Bakery	Singapore

whether it be in two strains or all strains. To do so, we used [5] to construct CDBGs with k-mer sizes in $\{25, 100, 500, 1000\}$. We chose these values because it is still not well understood what k values are appropriate for pan-genome analysis. The AFS algorithm parameters were selected to be minsup = 2, because we were interested in identifying syntenic blocks present in two or more strains, and $\epsilon_r = \epsilon_c = \mu_i = 0.10$, because we wanted there to be 90% identity among the supporting sequences of each block while allowing for insertions that are small relative to the size of the blocks. Generally, AFS algorithm parameters should be chosen relative to the desired level of conservation and prevalence of the blocks to be mined. Furthermore, the user should be mindful of the interplay between these parameters and the quality of the results. For example, if $\epsilon_r = \epsilon_c = \mu_i = 0$, then only exactly matching sequences will be identified. Conversely, if $\epsilon_r = \epsilon_c = \mu_i = 1$, then the relationship among

the sequences identified will be tenuous, at best. Although the legitimacy of the evolutionary relationships among the sequences identified should be determined by a domain expert, understanding that more lenient parameters increase the likelihood of false positives is important.

We mined copy number variants from the yeast CDBGs, specifically looking for regions that had expanded compared to other strains. Copy number expansions and contractions can have functional consequences, pointing to regions that are important in adaptation to the environment [23, 49, 110]. We looked in depth at the longest region — measured by the number of graph nodes — that had more than two copies in a single genome ($k = 1000$). In this case, there were three copies found in the Red Star assembly, which was derived from a commercial bakery yeast, and one copy found in the LalvinQA23 assembly, which is a wine yeast. We hypothesized that these regions would contain genes with important functions in bread making, given that its copy number is increased in Red Star. Each of the regions contained one full-length gene: *HFI1*. The HFI1 gene functions in chromatin modification, DNA damage repair and transcriptional regulation. This gene has, indeed, been found to be important in bread making. Specifically, it has a critical role in air-drying stress, which occurs during bread making [97] and is also required for anaerobic, but not aerobic, respiration. Anaerobic respiration is important in both bread making and alcohol generation [45].

The HFI1 AFS that we analyzed had representatives from only two species, yet the full-length HFI1 gene was found in 8 of the 10 yeast assemblies and a partial gene in an additional assembly. A Muscle multiple sequence analysis [27] and visualization in Jalview [12] (version 2.9.0b2) of the HFI1 gene shows only six variant sites of 1467 nucleotides (Figure 2.2).

The reason that only four HFI1 genes from two strains were put in

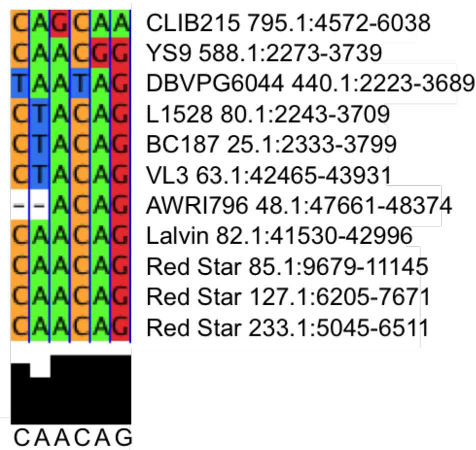

Figure 2.2: Muscle multiple sequence alignment of the HFI1 genes showing only the positions where differences occur. Yeast names are followed by scaffolds and nucleotide positions.

the AFS that we considered is likely due to the stringency of the $k = 1000$ constraint, which grouped the four identical HFI1 genes but excluded the others that differed at 1 or 2bp for a total of 6 polymorphic sites among the strains over the 1467bp of the HFI1 gene. The three Red Star and the LalvinQA23 HFI1 genes either represent very recent duplications, gene conversion, or transfer events, given their high sequence identity. Indeed, this is supported by a neighbor joining tree (generated within Jalview) for the HFI1 gene that shows no differentiation between these genes (Figure 2.3). Though these two strains do not share an obvious recent ancestor, geography, or industry that would explain the presence of identical copies of the HFI1 gene, active gene interchange has been suggested in yeast, especially in regard to genes with copy number differences between strains [24], though more work would be needed to explore this possibility.

We recognize that genome assemblies are not exact replicates of the genome from which they are derived because of sequencing bias and error, as well as artifacts in

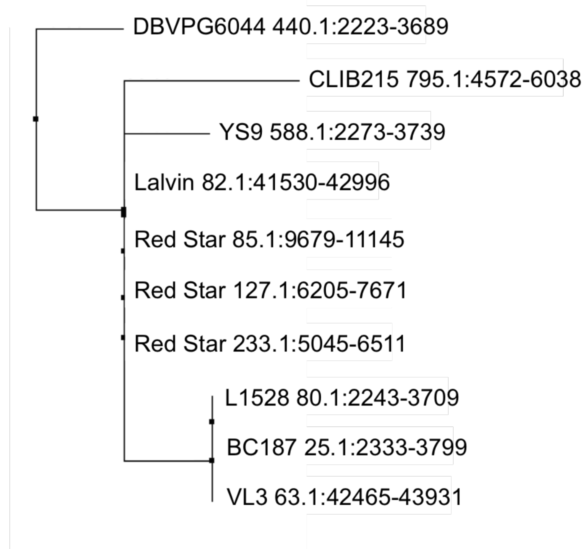

Figure 2.3: A phylogenetic tree of the full-length HFI1 genes. Strains that contain identical copies of the gene are located at the same levels in the tree, represented as branch points (dots) with zero depth.

the assembly, such as chimeric regions, regions that are missing from the assembly, and repetitive and/or duplicated regions that are over-collapsed, over-expanded, or simply over-corrected to make the different copies appear more similar than they are. Particularly relevant here is that, without experimental validation, we cannot confirm that the three Red Star HFI1 genes in the assembly match those in the genome, nor that we are not missing additional copies (or assembling extra copies) from Red Star or other strains. Nevertheless, our goal here is to develop a pan-genomics algorithm using the yeast assemblies as a model system rather than focusing on yeast pan-genomics, per se. As such, even with the shortcomings of the assemblies, we have demonstrated that biologically relevant information can be gleaned by applying our algorithm to multiple genomes from a single species, generating interesting biological hypotheses that can be followed up with experimental work.

2.7 Conclusions

A promising strength of pan-genomics algorithms and their ability to analyze multiple related assemblies simultaneously without reference bias is the ability to find patterns both of divergence (novelty) and conservation. Identification of divergent regions (regions that do not fall into AFSs even with lenient parameters) allows the detection of novel genes that are not present in the reference sequence and/or other assemblies from a species. These genes could represent genes obtained through horizontal transfer, hybridization, or strong positive selection, and may have important adaptive functions. Identification of regions that are conserved, on the other hand, allows determination of core gene sets that are required for the species. Identification of unannotated regions that are conserved across the species is also important. Conservation implies that purifying selection has been active to keep important regions conserved. Finding unannotated, conserved regions can lead to the identification of new genes or important regulatory elements.

Gene copy expansion and contraction, which we demonstrated above, can be an important mechanism of adaptation. Examining the AFS paths can differentiate between tandem duplications, which can be a response to environmental pressures that involved one or more genes, from duplications that resulted from segmental or whole genome duplications (polyploidy) that involve multiple genes. Identification of polyploidy-derived AFSs, whether auto-[4] or allo-polyploidy,[5] allows the detection of paralogous regions that can be flagged or merged, as desired. In addition, paralogous genes can be identified and their fate determined (gene loss, subfunctionalization, neofunctionalization). For allopolyploidy events, parental contribution and allelic

[4]Having two or more sets of chromosomes.

[5]Having three or more sets of chromosomes.

dosage can be determined.

In this work, we are concerned with pan-genomes at the nucleotide level, but pan-genomics is a powerful approach that can be applied at other levels as well. For instance, our algorithm can be applied on the nucleotide level or the amino acid level. It can further be applied at domain, gene, gene family, operon, or molecule (chromosome or plasmid) level. This flexibility allows researchers to tailor the algorithm to their questions and organisms, and to apply the algorithm across different evolutionary scales.

In the following Chapters, we consider a problem similar to the AFS problem for mining synteny from pan-genomic graphs in order to exploit more scalable algorithmic techniques.

CHAPTER THREE

FREQUENTED REGIONS

In this Chapter, we further address the problem of mining synteny, and subsequently research Questions 2 through 4 from Chapter 1, with an emphasis on Questions 2 and 3, by considering the problem of identifying regions within a pan-genome de Bruijn graph that are traversed by many sequence paths. We define such regions and the subpaths that traverse them as Frequented Regions (FRs).

3.1 Contributions

Expanding on Contribution 2 from Chapter 1, we formalize the FR problem, discuss its computational complexity, and describe effective and efficient, parallelized algorithms for finding FRs. Subsequently, we propose applications of FRs based on machine-learning (ranking of FRs via a probability density function and classification using Support Vector Machines) and pan-genome graph simplification. We demonstrate the effectiveness of these applications using data sets for the organisms *Staphylococcus aureus* (bacteria) and *Saccharomyces cerevisiae* (yeast). We corroborate the biological relevance of FRs by identifying insertions in yeast that aid in alcohol tolerance, and show that FRs are useful for classification of yeast strains by industrial origin, further demonstrating the efficacy of using path-graph methods as a means of analyzing pan-genomic data. Additionally, we present FR-based visualizations of pan-genomic space that use FRs to simplify pan-genomic graphs and the Canberra distance to perform multidimensional-scaling of strain origins by their FR content.

3.2 Introduction

As discussed in Chapter 2, an important problem in population genomics is the identification of synteny, that is, sequences that are (in)exactly preserved within the population. Though effective, the AFS problem presented in Chapter 2 posed a few challenges to devising algorithms, namely the requirement that the graph structures to be identified are subpaths (linear sequences of graph nodes). This could, for example, cause an algorithm to identify each supporting path of a frequent subpath as the anchor path for a different but effectively equivalent frequent subpath. This is a special case of the more general problem of two anchor paths that are practically identical, perhaps only differing by a single node, being identified as separate frequent subpaths. This causes redundancies in both computation and the result set. For these reasons, in this Chapter we present the Frequented Regions (FR) problem, which aims to alleviate these issues by mining regions, rather than subpaths, of a graph that are frequently traversed by a set of paths. By exploiting the structure of the graph we are able to develop an efficient algorithm that effectively mines inexact syntenic regions from pan-genomic graphs. Note, as we will see in Chapter 4, the AFS problem is still relevant.

3.3 Related Work

Given the FR problem's relatedness to the AFS problem, the related work is quite similar. Regardless, the contents of this section are thorough for the sake of completeness.

The FR problem is somewhat similar to other data mining problems, especially Frequent Itemset Mining (FI), which identifies sets of items that frequently occur together in a database of transactions. Specifically, the database is a binary matrix

where the columns correspond to items and the rows to transactions. If an item occurred in a transaction, then its cell has a 1, otherwise 0. An itemset is considered *frequent* if each of its items occurred together in *minsup* transactions, where minsup is either a fraction of the transactions in the database or a minimum number of transactions. Transactions in which a frequent itemset's items occur together are called *supporting transactions*, since they support the itemset as being frequent. A seminal FI algorithm, Apriori, was introduced in [1]. It works by constructing itemsets in a bottom-up, combinatorial manner and has an exponential run-time complexity. There are variations of FI that are tolerant to noise in the data [11, 64, 111]. These require that the supporting transactions in the transaction matrix meet a *row error threshold* constraint and/or the items meet a *column error threshold* constraint. Like exact FI algorithms, these do not consider the structure of the underlying graph and so are likely to generate several false positives.

Also related is Frequent Subgraph Mining, which is concerned with finding subgraphs that frequently occur in a database of graphs [42]. It is a well studied problem [47] commonly applied to biological data sets [36, 40, 53]. It could be applied to paths through a graph by treating each path as a different graph, but little work has been done with regards to discovering approximate solutions or scalability [62]. Another related graph problem is Exact Frequent Subpath Mining, which is described in [33]. The author shows that the problem of mining exact frequent subpaths is similar to FI, but differs in that the structure of the graph can be exploited to achieve more efficient running time. Unfortunately, the algorithm is incapable of mining frequent subpaths whose supporting paths contain some error. Furthermore, like Apriori, the algorithm works by constructing subpaths in a bottom-up, combinatorial manner and has exponential run-time complexity.

There exist a variety of tools for pan-genome analysis [106]. Unfortunately,

few of these are concerned with mining synteny [61] or scale beyond populations of microbial genomes, both of which are needs of the pan-genomics community [17]. One exception to the lack of synteny-based approaches is Sibelia [74], which determines syntenic regions from pan-genomes by iteratively eliminating bubbles in a de Bruijn graph with the sequence modification algorithm [89]. Since all bubbles are eventually merged into a single syntenic region, regions that are truly divergent will be falsely identified as syntenic, requiring the user to manually differentiate between false and true positives – a tedious task. Furthermore, this method also does not scale beyond populations of microbial genomes, as noted in Section 3.7.

3.4 Problem Definition

We assume the following input and parameters are supplied: A graph G and set of paths P within G. In our application, G corresponds to the compressed de Bruijn graph (CDBG) representing a pan-genome composed of multiple genomic sequences, where each sequence corresponds to a path p in G; P is the collection of these paths.

A *frequented region* (FR) is characterized by a set of nodes C and a set of supporting subpaths from P that pass through the nodes in C. There are two error parameters that we consider: 1) the minimum fraction of the nodes in C that each subpath must contain; we call this the *penetrance* parameter α, and 2) if a subpath from P leaves C, the number of steps to return to C (measured by the length of the corresponding sequence insertion); we call this the *maximum insertion* parameter κ. With this in mind, we formalize the definition of an FR as follows:

Given a path $p \in P$, let $p = \langle n_1, n_2, \ldots, n_L \rangle$, where n_i is the ith node visited by p and L is the length of the path. We define a subpath as $p[i, j] = \langle n_i, n_{i+1}, \ldots, n_j \rangle$ and $\text{seq}(p[i, j])$ as the genomic sequence corresponding to $p[i, j]$ in G.

Definition 2. *We say $p[i, j]$ is an (α, κ)-supporting subpath for a set of nodes C if and only if*

1. *$n_i, n_j \in C$ and between any two consecutive C nodes in $p[i, j]$, any gap of inserted sequence is at most κ in length; see Equation (3.3) for a computational definition.*

2. *$p[i, j]$ is* maximal *in the sense that it cannot be extended to either the left or right, or rather, $\nexists$ (α, κ)-supporting subpath $p[i', j']$ s.t. $[i, j] \subset [i', j']$.*

3. *$|p[i, j] \cap C| \geq \alpha|C|$.*

Note that (α, κ)-supporting subpaths do not overlap due to the maximality requirement; if they did overlap they could be merged. It is also fairly easy to identify all of the (α, κ)-supporting subpaths for a given path p and node set C; we just need to identify all maximal runs of C nodes in p whose corresponding sequences have insertions of length at most κ and then check if the run contains at least $\alpha|C|$ distinct nodes from C. Algorithm 7 in Section 3.5 implements this idea.

Definition 3. *A* frequented region *(FR) is a tuple (C, S), where C is a set of CDBG nodes and S is a set of (α, κ)-supporting subpaths of paths from P.*

We say C is the *node-set* and S is the *supporting-subpath-set* for the FR. We define the *support* of the FR as

$$\text{support}(C, S) = |S| \tag{3.1}$$

and define the *average length* of the FR as

$$\text{average-length}(\text{FR}) = \frac{\sum_{p[i,j] \in S} |\text{seq}(p[i, j])|}{|S|}. \tag{3.2}$$

Our goal is to find FRs that have high support. As such, an important point to consider is that, unlike FIs, FRs do not necessarily meet the Apriori Property [1], that is, sub-regions of a valid FR are not necessarily valid FRs themselves.

Lemma 3.4.1. *Sub-regions of a valid FR are not necessarily valid FRs themselves.*

Proof. This can be proven directly from the FR problem definition. Observe that every supporting subpath of a valid FR need not support every node in the FR, but rather an α fraction of the nodes. This means a supporting subpath may not necessarily support an α fraction of the nodes in some sub-region of the FR. Therefore, each supporting subpath of a valid FR may not necessarily support each of the FR's sub-regions, implying the sub-regions of the FR are not necessarily valid FRs themselves.

□

Conversely, if $\alpha = 0$, then FRs meet the Apriori Property.

Lemma 3.4.2. *If $\alpha = 1$, then sub-regions of a valid FR are themselves valid FRs.*

Proof. This can be proven directly from the FR problem definition. Observe, when $\alpha = 1$, then every supporting subpath of a valid FR must support every node in the FR. This means a supporting subpath must support each node in any valid sub-region of the FR. Therefore, each supporting subpath of a valid FR must support each of the FR's sub-regions, implying the sub-regions of the FR are themselves valid FRs. □

The computational problem considered is to find FRs that have high support and high average length.

Figure 3.1 provides an example.

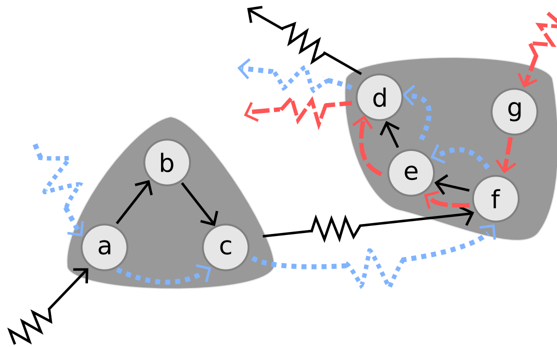

Figure 3.1: Example FRs. Assuming $\alpha \leq \frac{2}{3}$, the left group of nodes $C_L = \{a, b, c\}$ forms an FR with support 2 (from the black (solid) and blue (dotted) paths). The right group of nodes $C_R = \{d, e, f, g\}$ forms an FR with support 3 (from the black, blue, and red (dashed) paths). If C_L and C_R were merged, the merged FR would have support 2, provided the connecting black and blue path segments between nodes c and f each have at most κ insertions. Figure created by Brittany Fasy.

3.4.1 Problem Complexity

Counting exact Frequent Itemsets is known to be #P-complete [112]. It can be seen that the exact FI problem can be reduced to the FR problem.

Lemma 3.4.3. *Counting the number of FRs in a graph is #P-complete.*

Proof. The proof is reduction from the FI problem. Given an instance of the FI problem, (I, T), where I is a set of items and T is a set of transactions — each containing one or more items from I — create a connected graph G with a vertex for each item in I and a set of paths P containing a path for each transaction in T. The ordering of the vertices in each path in P does not matter. By Lemma 3.4.2, we let $\alpha = 1$ and $\kappa = \infty$ so that FRs in (G, P) meet the Apriori Property. By this reduction, there is a one-to-one correspondence between FIs in (I, T) and FRs in (G, P), so both problems have the same number of solutions – the reduction is parsimonious. This means any algorithm that can count the number of FRs in (G, P) can be used to count the number of Frequent Itemsets in (I, T), and vice versa.

Therefore, counting the number of FRs in a graph is #P-complete. □

Unlike the AFS problem in Chapter 2, deciding if there exists a set S of supporting paths for a given candidate node-set C is tractable. This is because there is no parameter that constrains the number of paths that must support each node in $C - \epsilon_c$ in the case of the AFS problem. In Section 3.5, we present an optimal algorithm for computing such sets of supporting paths.

3.5 Algorithm

Due to the vast scale of pan-genomic data, we opted for a simple heuristic approach to find interesting FRs. The basic idea of the algorithm is to find FRs in an agglomerative (bottom-up) fashion, where each CDBG node starts in its own cluster and pairs of clusters are repeatedly merged, forming a hierarchy of FRs. When an FR is created for a set of nodes C, the corresponding set of supporting subpaths S must be found. This is accomplished by the ComputeSupport subroutine shown as Algorithm 3.1. Note, in line 7, the `gap` subroutine returns the length of the inserted sequence between the consecutive matching node $m[i]$ and $m[i+1]$ along the path p; each end of $p[m[i], m[i+1]]$ matches a node in C, so the insert gap length is

$$\text{gap}(p, m, i) \;=\; |\text{seq}(p[m[i], m[i+1]])| - |\text{seq}(p[m[i]])| - |\text{seq}(p[m[i+1]])|. \quad (3.3)$$

The idea of the algorithm is to find a pair of existing FRs and merge them such that the newly created FR has the greatest possible support. Specifically, the result of every possible merge (one for each inter-cluster edge) is computed in turn with Algorithm 3.2, which unions the node sets of the two FRs being merged, C_L and C_R, and then computes the corresponding set of supporting subpaths S with

Algorithm 3.1: Compute FR supporting path segments.

```
1: function ComputeSupport(C,p)
2:     Let S = ∅
3:     Let m = [i : p[i] ∈ C]
4:     start = 1
5:     while start ≤ |m| do
6:         i = start
7:         while i < |m|, gap(p, m, i) ≤ κ do
8:             i = i + 1
9:         end while
10:        if (i − start + 1) ≥ α|C| then
11:            S = S ∪ p[m[start], m[i]]
12:        end if
13:        start = i
14:    end while
15:    return S
16: end function
```

Algorithm 3.1. The merge that would yield the FR with the highest support is then performed. The procedure terminates when no merge will yield an FR with positive support. Pseudocode of the procedure is provided in Algorithm 3.3. Line 10 can be computed efficiently using a priority queue of possible merges. When a new FR is formed by merging (C_L, S_L) and (C_R, S_R), other potential merges involving either of these FRs remaining in the queue must be updated to be potential merges with the newly formed FR. Note that S_L and S_R are not actually used by Algorithm 3.2 since a new supporting set must be computed in order to capture supporting subpaths that

Algorithm 3.2: Evaluate FR merge.

1: **function** EVALMERGE$((C_L, S_L), (C_R, S_R))$
2: Let $C = C_L \cup C_R$.
3: Let $S = \emptyset$.
4: **for** $p \in P$ **do**
5: $S = S\cup$ COMPUTESUPPORT(C, p)
6: **end for**
7: **return** S
8: **end function**

may not have been present in S_L or S_R.

After Algorithm 3.3 completes, the FRs will form a *hierarchical clustering* with one or more root FRs, that is, FRs that are not contained within another FR. The final step of the algorithm is to filter the FRs with high support. A simple recursive procedure is used to do this, as outlined in Algorithm 3.4. For each root FR (C, S) found, we call EXPLORE$((C, S, 1))$. This will recursively explore the tree in a depth-first manner and *report* the FR associated with a tree node t if and only if t's support is greater than any of its ancestors (superset FRs). All such reported FRs are considered *interesting* FRs (iFRs). All reported FRs are interesting, so *reporting* an FR (**report**$((C, S))$ on Line 3) means bringing it to the user's attention, whether by writing it to a file or saving it to a data structure for additional analysis. We limit the analyses in Section 3.7 to such FRs.

3.5.1 Reverse-Complement Support

Genes and other relevant sequence features can be encoded in either the forward or reverse-complement direction on a particular DNA sequence. We adopt a simple

Algorithm 3.3: Agglomerate FRs.

1: **procedure** MergeFRs(G, P)
2: **for** $n \in \text{nodes}(G)$ **do**
3: Let $S = \emptyset$.
4: **for** $p \in P$ **do**
5: $S = S \cup$ ComputeSupport($\{n\}, p$)
6: **end for**
7: Create FR ($\{n\}, S$).
8: **end for**
9: **repeat**
10:

$$(C_L, S_L), (C_R, S_R) = \underset{(C_L,S_L),(C_R,S_R)}{\text{argmax}} |\text{EvalMerge}((C_L, S_L), (C_R, S_R))|$$

11: Let $C = C_L \cup C_R$.
12: Let $S = \text{EvalMerge}((C_L, S_L), (C_R, S_R))$
13: Create FR (C, S)
14: Mark (C_L, S_L), (C_R, S_R) unavailable for subsequent mergers
15: **until** $\text{EvalMerge}((C_L, S_L), (C_R, S_R)) = \emptyset$
16: **end procedure**

approach for detecting sequences that support a particular FR in the reverse-complement direction: For each sequence in the data set we generate a path p in the forward direction and a corresponding path $\bar{p}^r$ in the reverse-complement direction. This permits FRs to have supporting subpaths from either p or $\bar{p}^r$. Observe that

Algorithm 3.4: Report interesting FRs.

1: **procedure** EXPLORE$((C,S),m)$
2: **if** support$(C,S) \geq m$ **then**
3: **report** (C,S).
4: **end if**
5: Suppose $(C,S) = \text{merge}(C_L, S_L, C_R, S_R)$
6: Let $m' = \max(m, \text{support}(C,S)+1)$
7: EXPLORE$((C_L, S_L), m')$
8: EXPLORE$((C_R, S_R), m')$
9: **end procedure**

for any FR (C,S), there is a reverse-complement version of it $(\bar{C}^r, \bar{S}^r)$, in which the FR nodes and supporting subpaths are reverse-complemented. To reduce the number of iFRs reported, we only report FRs with supporting path sets that are comprised of at least 50% paths in the forward direction. If an FR fails to be reported, there is a corresponding reverse-complemented FR that will be. This approach can be implemented by adding each genome's reverse complement to the set of genome sequences prior to constructing the population's CDBG. Our `FindFRs` software described in Section 3.7 implements this approach and allows users to specify if reverse-complemented paths should be considered.

3.5.2 Weighted Support

In Algorithm 3.3, FR merges are performed greedily in most-support-first order. We note that this metric can be weighted so that additional FR properties are emphasized during the selection process. For example, the total support of each potential merge can be weighted by the average penetrance ($\geq \alpha$) of the

supporting paths. This can prevent otherwise coherent FRs from being involved in tenuous merges due to lenient parameterization. Other weights could be based on insertion length, the size of the FRs being merged, or even classification labels associated with the sequences.

3.5.3 Parallelization

The main loop in Algorithm 3.2 is trivially parallelizable, as computing the supporting subpaths for any path $p \in P$ is independent of the other paths. It is also possible to parallelize the main loop of Algorithm 3.3. We observe that merging FRs can be done consistently in parallel, provided no FR is involved in more than one merge. This is equivalent to the requirement that edges representing the FR pairs chosen for merging form a *matching*. To parallelize Algorithm 3.3, we first compute the score of each merger edge, using Algorithm 3.2, which can be done in parallel since each merger edge's score is independent of the others. We compute a *maximal weighted matching*, which will represent the FR mergers to be done in parallel. The matching is *maximal*, rather than a *maximum*, because we still want to greedily select edges with higher weights.

Finding maximal weighted matchings is a well-studied problem [28] for which there exist fast parallel [3] and distributed algorithms [38]. In this work, we consider two maximal matching algorithms: 1) A simple linear sorting based algorithm that greedily selects edges in greatest-weight-first order[1] [90] (Algorithm 3.5), and 2) a parallelized version of the linear algorithm based on locally dominant edges [70] (Algorithm 3.6). Both algorithms achieve a $\frac{1}{2}$-approximation of the optimal maximum weight matching.

Both algorithms utilize a map π to report what vertex, if any, each vertex is

[1] We use this as a baseline to measure the speedup of our parallel implementation.

Algorithm 3.5: Linear maximal weighted graph matching.

```
1: function LinearMatching(V, E)
2:     for v ∈ V do
3:         π(v) = ∞
4:     end for
5:     for {u, v} ∈ E in order of descending weight do
6:         if π(u) == ∞ and π(v) == ∞ then
7:             π(u) = v
8:             π(v) = u
9:         end if
10:    end for
11: end function
```

matched to. Specifically, Algorithm 3.5 works by initializing each vertex's entry in π to infinity (Line 3). It then iterates the edges in greatest-weight-first order, adding edges to the matching only if both its vertices' π values are infinity (Lines 5-10). In order to achieve parallelism, Algorithm 3.6 assumes each vertex v maintains a list of its neighbors in greatest-weight-first order, denoted v.neighbors. The algorithm works by iterating until the edge set E is empty. Each iteration begins by setting each vertex's π value to its most weighted neighbor, v.neighbors.first, in parallel (Lines 3-5). It then iterates the vertices in parallel, checking if each vertex's π value's π value is equal to itself – a matching. If so, it removes the corresponding edge from the edge set and itself from its neighbors' neighbor lists, otherwise, the vertex's π value is set to infinity (Lines 6-15 and 18-23).

Algorithm 3.6: Parallel maximal weighted graph matching.

```
1: function ParallelMatching(V, E)
2:     while E ≠ ∅ do
3:         for v ∈ V do parallel
4:             π(v) = v.neighbors.first
5:         end for
6:         for v ∈ V do parallel
7:             u = π(v)
8:             if π(u) == v then
9:                 E = E \ {v, u}
10:                RemoveNode(E, v)
11:                RemoveNode(E, u)
12:            else
13:                π(v) = ∞
14:            end if
15:        end for
16:    end while
17: end function
18: function RemoveNode(E, v)
19:    for u ∈ v.neighbors do
20:        u.neighbors = u.neighbors \ v
21:        E = E \ {v, u}
22:    end for
23: end function
```

3.5.4 Time Complexity

Suppose the CDBG contains V vertices and E edges and let L be the total length of all the pan-genomic paths in P. Algorithm 3.3 is the main program; it begins with creating a new FR for each node $n \in G$ and finds its support. This can be done in $O(V + L)$ time. Next, we note that there are at most $V - 1$ iterations of the repeat-until loop since each iteration creates an internal node in a binary tree with V leaves. Determining the next FR merger (internal node) can be done efficiently using a priority queue. After each FR merger, the queue needs to be updated. Observe that $E = O(V)$, since at the start of the algorithm each vertex has $O(1)$ incident edges and E can only decrease as merges happen. This means the queue can be updated in $O(V + L + V \lg V)$ time, where $O(V + L)$ accounts for computing the support of all new potential FR mergers with the newly-created FR. The overall time complexity is thus, $O(LV + V^2 \lg V)$.

Algorithm 3.5 computes a matching in $O(E \lg E)$ time, since it requires the edges be sorted by greatest weight. On average, the number of FRs merged at each iteration is half, meaning it takes $O(\lg V)$ iterations to construct the FR hierarchy on average, and $O(V)$ iterations in the worst case. Accounting for the computation of support and the decreasing number of edges at each iteration, this results in an average run-time complexity of $O(L \lg V + V \lg V)$, and a worst case complexity $O(LV + V \lg V)$.

In Algorithm 3.6, if the edge weights are distributed randomly, then the main loop is expected to terminate after $O(\lg E)$ iterations, though the worst case is $O(E)$. By representing each vertex's neighbor list with an efficient data structure, such as a linked list, the time complexity of all the vertex operations is dominated by the initial sorting of the vertex neighbors. Specifically, the accumulated work performed on all vertices is $\sum_{v \in V} O(\delta(v) \lg \delta(v)) = O(E \lg \Delta)$, where $\delta(v)$ is the degree of vertex v and Δ is the maximum degree in the graph. Since all vertices are iterated at each

iteration of the algorithm, this gives an expected time complexity of $O(V \lg E)$ and a worst case complexity of $O(VE)$ [70]. Thus, the overall expected time complexity is $O(L \lg V + V \lg^2 V)$, and the worst case complexity is $O(LV + V^2 \lg V)$.

3.6 Applications

In this section we discuss some applications for FRs including machine learning methods and an approach for pan-genome graph simplification and visualization.

3.6.1 Machine Learning with FRs

Like other sequence features, such as SNPs, FRs may provide sequence characteristics that can be used to distinguish or categorize groups of genomes. For example, in Section 3.7 we differentiate between yeast strain genomes based on their industrial origin. In order to evaluate the effectiveness of FRs for this task we model this as a *multi-class classification* problem in which each strain is annotated with one of the industrial-origin class labels, and each iFR is a feature. The problem is then to apply a *supervised learning* algorithm to the feature set, or a subset of the feature set, such that unlabeled strains are labeled with the correct class.

Support Vector Machines (SVMs) [19] have been shown to be an effective approach to classifying genomes based on shared genetic features. Traditionally, such classifications are done with Genome Wide Association Studies (GWAS) and/or Principal Component Analysis (PCA) [94], however, SVMs have been shown to have more classification power than GWAS/PCA [10]. Therefore, in this work we use SVMs for classifying genomes within a pan-genome based on their FR content, as described below.

Specifically, each example (i.e. genome) is represented as an n-dimensional vector (n is the total number of FRs used) in which each individual component of

a vector corresponds to the number of times a certain FR occurs with that genome, similar to a bag-of-words model [48, 72]. We use these vectors as input to our SVM model [82] and evaluate its accuracy in predicting the correct industrial origin based on the FRs they are associated with (see Section 3.7.2.3 for results of this study).

However, depending on the parameters used and size of the pan-genome, a large number of iFRs may be identified by Algorithm 3.4. These may span a variety of classes, ranging from the pan to strain-specific, among others. As such, *feature selection* is an important task when trying to maximize classification power. An effective feature selection strategy is one that can identify a small number of features that can discriminate classes based on their attributes.

A simple approach based on multinomial distributions which we refer to as *multinomial-filter* is as follows: Suppose that the sequence paths are divided into a set of groups $\{G_1, \ldots, G_k\}$. Let

$$c_{ij} = \sum_{p \in G_i} \text{support}_p(\text{FR}_j), \tag{3.4}$$

be the total support of FR_j for all sequence paths belonging to group G_i. In other words, c_{ij} is a count of the number of times FR_j occurs in G_i. Let $T_i = \sum_j c_{ij}$ be the total count of iFRs found in group G_i and let $T = \sum_i T_i$ be the total across all groups. The frequency with which a random iFR occurs in G_i is then $f_i = {}^{T_i + 1}/_{T}$. A pseudo-count of 1 is added to the count for each group to indicate that if an iFR is not observed in the group, then observing it is highly improbable, but not impossible. Generally, adding a pseudo-count in this manner is known as Laplace smoothing [95], and can be used to change the expected probability based on prior knowledge, such as the distribution of industrial-origin class labels across strains of yeast. In this study, the only prior knowledge we assume is that it is possible for any iFR to appear in

any group, so we use a pseudo-count of 1.

The probability of observing the group counts for FR_j in a random FR is multinomially distributed with bin probabilities $\langle f_i \rangle$,

$$\Pr(\text{FR}_j) = \frac{n!}{c_{1j}! \cdots c_{kj}!} f_1^{c_{1j}} \cdots f_k^{c_{kj}}, \tag{3.5}$$

where $n = \sum_i c_{ij}$. The lower the $\Pr(\text{FR}_j)$ value, the less likely that the observed group counts for that FR occurred by random chance. We note that $\Pr(\text{FR}_j)$ is similar but not identical to a p-value, as the latter includes the probability of at least as extreme data under the null hypothesis.

3.6.2 Graph Simplification

The goal of visualizing pan-gnomic data is to provide researchers with an intuitive and informative representation that can help expedite knowledge discovery. Visualizing a pan-genome graph is a difficult task, especially when dealing with larger, complex genomes. It is relatively simpler to visualize pan-genomes at the microbial level and with a limited number of genomes, but when the number of genomes and size increases, the result is an indecipherable "hairball", as depicted in the SplitMEM [71] work. To that end, we would like to use iFRs to create visualizations of pan-genomes that are human parsable, meaningful and facilitate knowledge discovery.

One approach is to filter what contents of a pan-genome are visualized by selecting a group of FRs and restricting attention to how the pan-genome traverses the corresponding FR node clusters. If we are given a list $L = \{C_1, \dots C_n\}$ of disjoint FR clusters, we can create a corresponding graph G_L with n vertices, where each vertex represents one of the FR clusters in L. For each sequence path p in the original CDBG, we can trace a path in G_L according to the order in which p supports the clusters in L. Figure 3.4 shows an application of the approach to visualize paths in a

yeast pan-genome.

3.7 Experimental Results

The FR-finding algorithm was implemented in Java. We will refer to these implementations as `FindFRs` and distinguish between them when necessary. We evaluated `FindFRs` on two data sets, *Staphylococcus aureus* and *Saccharomyces cerevisiae*. The *Staphylococcus aureus* data set was used to compare against Sibelia [74], the only other program currently available for mining synteny from a pan-genome de Bruijn graph. The *Saccharomyces cerevisiae* data set was used to illustrate the scalability of the `FindFRs` algorithm and exhibit a variety of FR-based analyses.

We used [5] to construct the CDBGs for each data set. Besides the main parameters α and κ, we also include two other parameters, *minsup* and *minsize*, which serve to limit the amount of output generated; only iFRs whose support and size (number of CDBG nodes) are at least these minimums are reported.

Experiments were run on a server with 4 Intel Xeon 2.2 GHz 32 core processors and 1 TB of RAM, however, most data sets could also run on standard desktop PCs, with longer running times.

As we discussed with the AFS algorithm parameters in Chapter 2, `FindFRs` parameters should be chosen relative to the desired level of conservation and prevalence of the synteny blocks to be mined. Furthermore, the user should be mindful of the interplay between these parameters and the quality of the results. For example, if $\alpha = 1$ and $\kappa = 0$, then only sequences that traverse the exact same CDBG vertices will be identified. Given a sufficiently large minsize parameter value, most all of an FR's sequences will be the same. Conversely, if $\alpha = 0$ and $\kappa = |V|$, where V is the set of vertices in the CDBG, then the relationship among the sequences identified will be

tenuous, at best. Although the legitimacy of the evolutionary relationships among the sequences identified should be determined by a domain expert, understanding that more lenient parameters increase the likelihood of false positives is important.

3.7.1 *Staphylococcus aureus*

In this section, we compare `FindFRs` to `Sibelia` [74]. We selected `Sibelia` because it is the only other tool for finding synteny blocks in pan-genomic de Bruijn graphs. We used the same four *Staphylococcus aureus* strains (JH1, N315, TW20, and MSSA476) that were used in the `Sibelia` paper for comparison. Both programs were run with k-mer sizes in $\{25, 100, 500, 1000\}$. We chose these values because it is still not well understood what k values are appropriate for pan-genome analysis. Two versions of `FindFRs` were used, one that performs merges based on FR support and one that performs merges based on FR support weighted by the average penetrance of the supporting paths, as described in Section 3.5.2. `Sibelia` was run with its default parameters. The `FindFRs` parameters were selected to be minsup $= 2$, because we were interested in identifying syntenic blocks present in two or more strains – the same as `Sibelia`; $\alpha = 0.5$, because it is ambiguous what sequence identity the `Sibelia` blocks would have, so 50% seemed neither too stringent or lenient; $\kappa = 0$, because `Sibelia` does not explicitly allow for insertions; and minsize $= 2$, because we did not want to identify trivial blocks (single de Bruijn nodes). The results of this experiment are shown in Table 3.1.

As we can see, `FindFRs` reports a higher number of frequented regions with and without weighted support compared to `Sibelias` synteny blocks, indicating the finer scale of partitioning regions based on biological significance. For the non-weighted test case, we estimated percent overlap between regions reported by these two algorithms: For all k-mer values, approximately 98% of `FindFRs` FRs overlapped with `Sibelia`'s

Table 3.1: `Sibelia` (using all default parameters, except for k values) versus `FindFRs` (using parameters $\alpha = 0.5$, $\kappa = 0$, minsup $= 2$, and minsize $= 2$) on 4 strains of *Staphylococcus aureus*. "*iFRs" and "*iFR roots" denote the results of `FindFRs` using support weighted average penetrance of the supporting paths.

Result type	$k = 25$	$k = 100$	$k = 500$	$k = 1000$
Synteny blocks	142	143	132	132
iFRs	10,740	1,334	2,011	1,351
*iFRs	172	461	1,064	383
iFR roots	392	122	97	134
*iFR roots	50	22	180	101

synteny blocks. Furthermore, we also computed overlap in terms of sequence percentage (i.e. what percentage of the `Sibelia` block sequences are contained in `FindFRs` FRs, and vice versa). For $k = 25$, alignments showed 91% of `Sibelia` Synteny block sequence base-pairs were contained within `FindFRs` frequented regions and only 82% of `FindFRs` frequented regions sequence base-pairs were contained within `Sibelia` Synteny blocks, indicating `FindFRs` FRs captured most of `Sibelia` Synteny blocks at the nucleotide base-pair level.

Lastly, to compare the run times of both programs, we tested a slightly larger data set consisting of 31 *Staphylococcus aureus* ($\approx$ 90Mb) strains. For $k = 25$, `Sibelia` took 186 minutes, whereas `FindFRs` took 43 minutes.

3.7.2 *Saccharomyces cerevisiae*

Saccharomyces cerevisiae (yeast) is a well studied model system with some published comparative genomic and pan-genomic work [8, 24], allowing us to use the existing knowledge base to assess the quality of the FRs found by our algorithm. Furthermore, yeast is highly diverse, economically important, and its multiple

industrial applications enable interesting functional genomics. These attributes, compounded with a genome size of approximately 12Mb, make yeast an interesting and tractable data set for algorithm testing.

Our yeast pan-genome was built with assemblies from the *Saccharomyces* Genome Database.[2] The data set consisted of 55 assemblies from 48 yeast strains over a wide range of industrial applications and geographic origins. The complete data set if described in Table 3.2.

We constructed yeast pan-genome CDBGs for k-mer sizes in $\{25, 100, 500, 1000\}$. Again, we chose these values because it is still not well understood what k values are appropriate for pan-genome analysis. Here, no weighted support was used, and we varied the α and κ parameters, as indicated in the various experiments. As a sample run, we ran `FindFRs` with parameter values $\alpha = .0.7$, $\kappa \in \{0, 3000\}$, minsup $= 5$, and minsize $= 5$. Table 3.3 indicates the size of the CDBGs created and the number of iFRs found ($\kappa = 0$ only). Figure 3.2 shows support versus average length. We note that `Sibelia` was not able to process this data set ($\approx$ 600Mb), so we do not report a comparison.

3.7.2.1 Parallelization Speedup We implemented both the linear and parallel maximal matching algorithms, Algorithm 3.5 and Algorithm 3.6, respectively. Since the number of paths in our data sets are relatively small, the computation of the maximal matchings dominates the run-time complexity. As such, here we compare the run-time of the linear and parallel versions of `FindFRs` in terms of the matching computed during the first iteration of the main loop in Algorithm 3.3, which is the largest graph for which a matching is computed during the run-time of `FindFRs`. The results are shown in Figure 3.3.

[2] `http://www.yeastgenome.org/`

Table 3.2: The 48 yeast strains with usage and region listed where known.

Strain	Source	Region	Strain	Source	Region	Strain	Source	Region
AWRI1631	Wine	South Africa	Foster's O	Ale		UC5	Sake	Japan
AWRI796	Wine	South Africa	FY1679	Laboratory		UWOPS05_217_3	Fruit/Nectar	Malaysia
BC187	Wine	California	JAY291	Bioethanol	Brazil	VIN13	Wine	South Africa
BY4741	Laboratory		JK9-3d	Laboratory		VL3	Wine	France
BY4742	Laboratory		K11	Sake	Japan	W303	Laboratory	
CBS7960	Bioethanol	Brazil	Kyokai7	Sake	Japan	X2180-1A	Laboratory	
CEN.PK	Laboratory		L1528	Wine	Chile	Y10	Fruit/Nectar	Coconut
CLIB215	Bakery	New Zealand	LalvinQA23	Wine	Portugal	Y55	Laboratory	
CLIB324	Bakery	Vietnam	M22	Fruit/Nectar	Italy	YJM269	Wine	Austria
CLIB382	Ale	Ireland	PW5	Wine	Nigeria	YJM339	Pathogen	
D273-10B	Laboratory		Red Star	Bakery		YJM789	Pathogen	
DBVPG6044	Wine	West African	RM11-1a	Fruit/Nectar	California	YPH499	Laboratory	
EC1118	Wine		SEY6210	Laboratory		YPS128	Nature	Pennsylvania
EC9-8	Nature	Israel	SK1	Laboratory		YPS163	Nature	Pennsylvania
FL100	Laboratory		T7	Nature	Missouri	YS9	Bakery	Singapore
Foster's B	Ale		T73	Wine	Spain	ZTW1	Bioethanol	China

Table 3.3: The size of each yeast CDBG (number of nodes and edges) and the number of iFRs found using `FindFRs` (using parameters $\alpha = .0.7$, $\kappa = 0$, minsup $= 5$, and minsize $= 5$).

k	CDBG nodes	CDBG edges	iFRs
25	2,260,767	38,390,883	115,585
100	1,758,760	21,320,923	97,550
500	890,055	5,036,766	29,766
1000	443,764	1,653,332	8,994

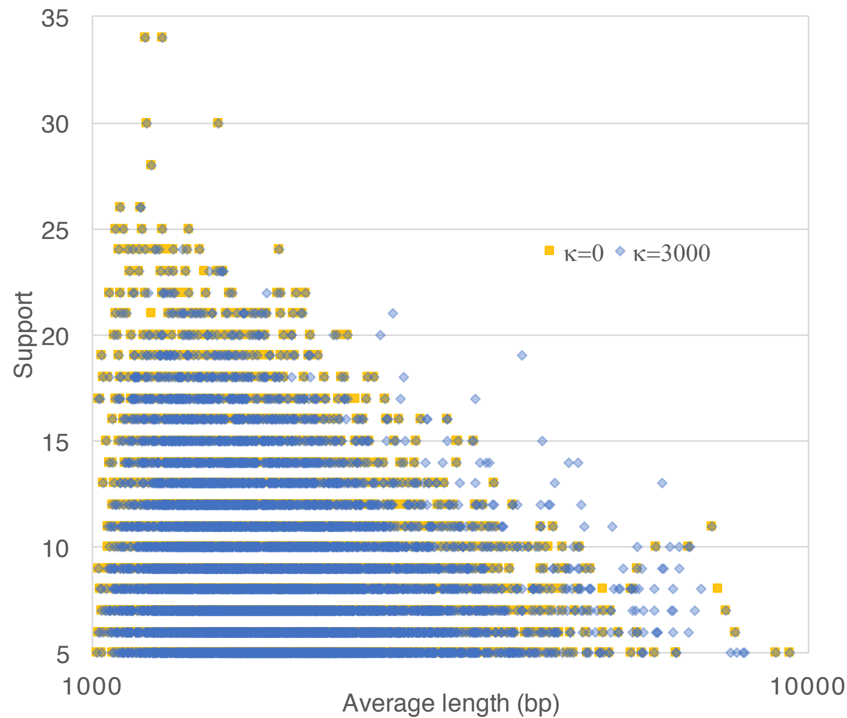

Figure 3.2: Distribution of yeast iFR support versus average length for FRs found with `FindFRs` (using parameters $\alpha = .0.7$, $\kappa \in \{0, 3000\}$, minsup $= 5$, and minsize $= 5$). Allowing insertions ($\kappa = 3000$) creates some longer FRs.

As we can see, when run with a single thread, Algorithm 3.6 has similar run-times as Algorithm 3.5. This indicates that Algorithm 3.6 is indeed running in the

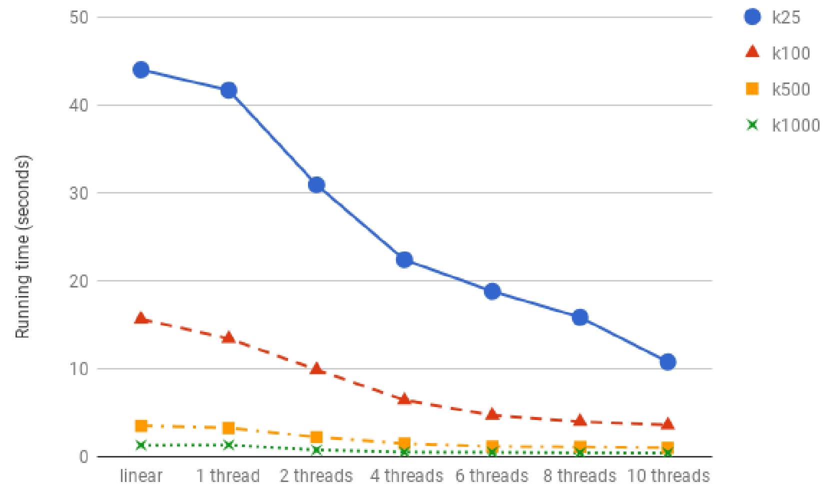

Figure 3.3: Running times of the linear and parallel graph maximal matching algorithms — Algorithm 3.5 and Algorithm 3.6, respectively — on the *Saccharomyces cerevisiae* CDBGs (Table 3.3).

expected amount of time, rather than the theoretical worst case. We also see that the total speed-up and the point at which the number of threads yields diminishing returns is proportional to the size of the graph – a similar result to existing speedup studies [3, 70]. This suggests that our method can scale to much larger pan-genomic graphs than considered here.

3.7.2.2 Consistency with Yeast Biology Because yeast is one of the simplest eukaryotes, it has been an important model system for genomic research [8, 24]. Yeast is also economically important; it is used in bread, biofuels, and alcoholic beverages, including wine, sake, and ale. Further, yeast is important in other contexts, being found in natural environmental systems, including as pathogens of humans.

The ability to thrive in harsh environments, which is a required attribute for industrial specialization in yeast, is often obtained through horizontal transfer of sets of genes [8, 24]. To determine whether we could uncover some of these novel

inserted regions, we looked at novel insertions that have been validated in the EC1118 yeast strain. These insertions introduce genes that allow EC1118, which is used in wine-making, to grow in the presence of alcohol. EC1118 has three novel insertions compared to S288C [84]. While S228C is not in our data set, we compared to laboratory strains that were derived from S288C (BY4741, BY4742, FY1679, X2180-1A, and YPH499). All three insertions were uncovered using our FR approach (Figure 3.4).

The shortest insertion is a 17kb novel insertion on chromosome XIV [84]. Five novel genes are inserted near the telomere between S288C genes YNL037C and YNL038W. The 17kb novel region was found on EC1118 sequence accession FN393084.1 ($k = 500$, $\alpha = 0.7$), which contained the insertion and anchoring sequence on both sides (Figure 3.4A). Multiple alcohol-related strains have versions of this insertion as indicated by the thicker alcohol edges as well as alcohol edges that follow alternate paths through the region. None of the laboratory strains, including the five S288C-derived strains, show support of any of the inserted FRs. Most of the nodes in the insertion occur only in alcohol-related strains, confirming that this region could be important in determining alcohol tolerance (Figure 3.4A).

The 38kb EC1118 insertion on one of the telomeres of chromosome VI [84] was also uncovered in EC1118 ($k = 500$, $\alpha = 0.7$, $\text{minsup} = 4$, $\text{minsize} = 4$; Figure 3.4B). This is a more complicated insertion that includes some rearrangements. A 23kb segment of chromosome VI was deleted in EC1118 while an adjacent segment was translocated to one of EC1118's chromosome X telomeres. The 38kb novel insertion merged onto the end of the chromosome, connecting to a 12kb segment translocated from EC1118's chromosome VIII, which merged with the remaining sequence from chromosome VI.

Many of the EC1118 FRs that were translocated to chromosome VI from

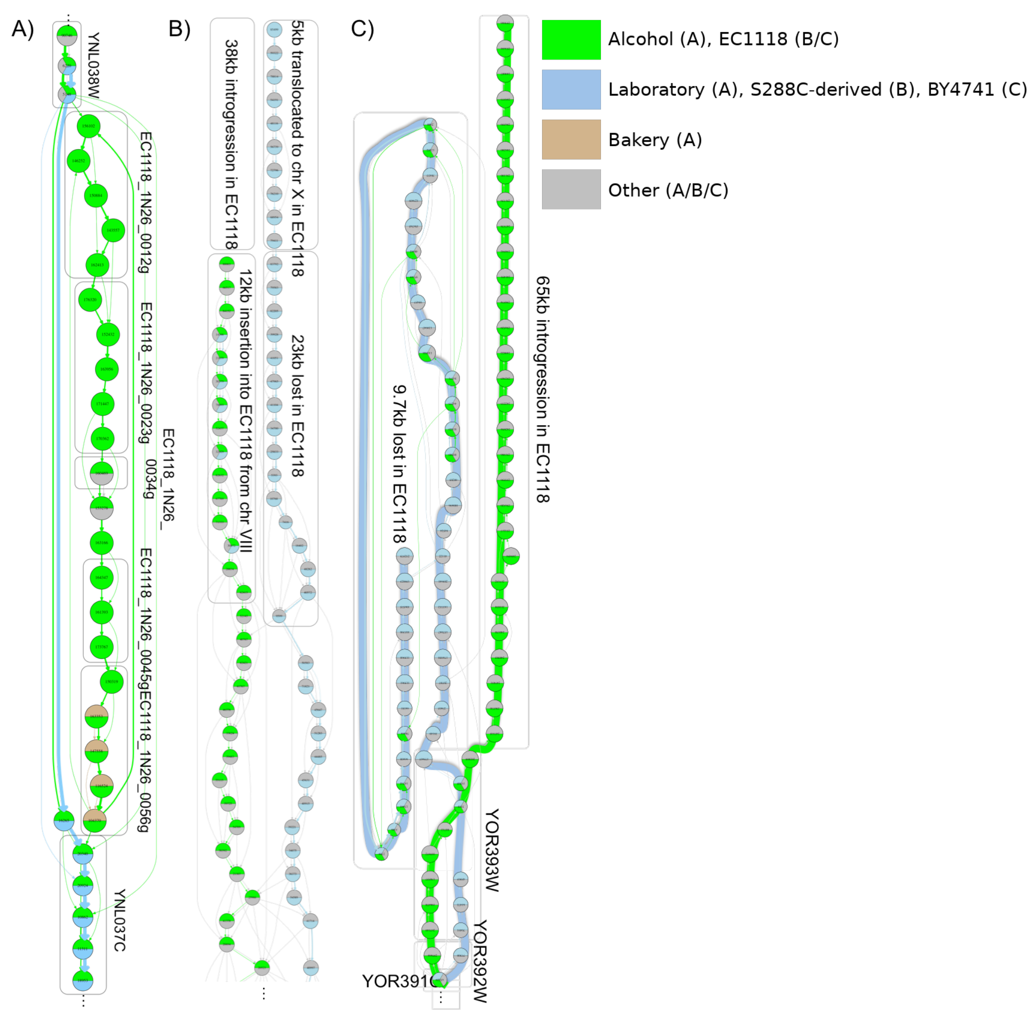

Figure 3.4: The three novel insertion regions [84] showing FRs from EC1118 compared to S288C (represented by the S288C-derived strain BY4741). (A) 17kb insertion on chromosome XIV showing alcohol (including wine, sake, ale and bioethanol strains), laboratory, bakery, and other strains. (B) A complex insertion event on chromosome VI shows a 12kb translocation in EC1118 from chromosome VIII, a 23kb deletion from in EC1118 and a 5kb region from S288C that is translocated to chromosome X in EC1118. Strains included were EC1118, S288C-derived (strains BY4741, BY4742, FY1679, X2180-1A, and YPH499), and other. (C) A 65kb novel insertion replaced 9.7kb of the chromosome XV telomere in EC1118. Shown are EC1118 and BY4741.

chromosome VIII (12kb) are shared by the S288C-derived accessions. This is not surprising because the S288C-derived accessions have this region on chromosome VIII.

As expected, none of the S288C-derived edges connect the 12kb chromosome VIII translocation with the rest of chromosome VIII. None of the FRs in the 23kb segment lost in EC1118 show support in EC1118, as expected given that it has been deleted in EC1118. However, none of the FRs in the 5kb region that was translocated to chromosome X in EC1118 are contained in EC1118, either, even though this region occurs on chromosome X. Presumably, it has evolved away from the S288C version to make it different enough that our parameters did not link the two regions into the same set of nodes. The EC1118 FRs stop approximately 38kb from the end of chromosome VI. No FRs were found in the 38kb novel region, presumably because there was not enough support among yeast strains to generate FRs. This raises the issue that sometimes it is interesting to look for regions where FRs are absent as they are potentially novel.

The final novel insertion is a 65kb region that replaced the last 9.7kb (including genes YOR394-C-A (2 copies), YOR394C, and YOR396W, not shown) of chromosome XV [84]. For simplicity, only EC1118 and BY4741 compared to all others are shown in Figure 3.4C and their chromosome XV paths have been highlighted ($k = 500$, $\alpha = 0.7$). The last gene shared by EC1118 and BY4741, YOR393W, has some shared FRs and some FRs that have diverged. Thereafter, the paths diverge. The EC1118 65kb novel insertion, shared with other yeast strains, is highly conserved as there are not alternate paths through the region. The 9.7kb region in BY4741, which was deleted in EC1118 actually has FRs that are shared with EC1118, though EC1118 does not have them on chromosome XV. These are telomeric genes that tend to occur on several yeast telomeres.

Our pan-genomics approach allowed us to quickly go beyond confirming EC1118's inserted regions to exploring these regions across multiple yeast genomes. As an example, we mined our graph for FRs from the 17kb EC1118 insertion in other

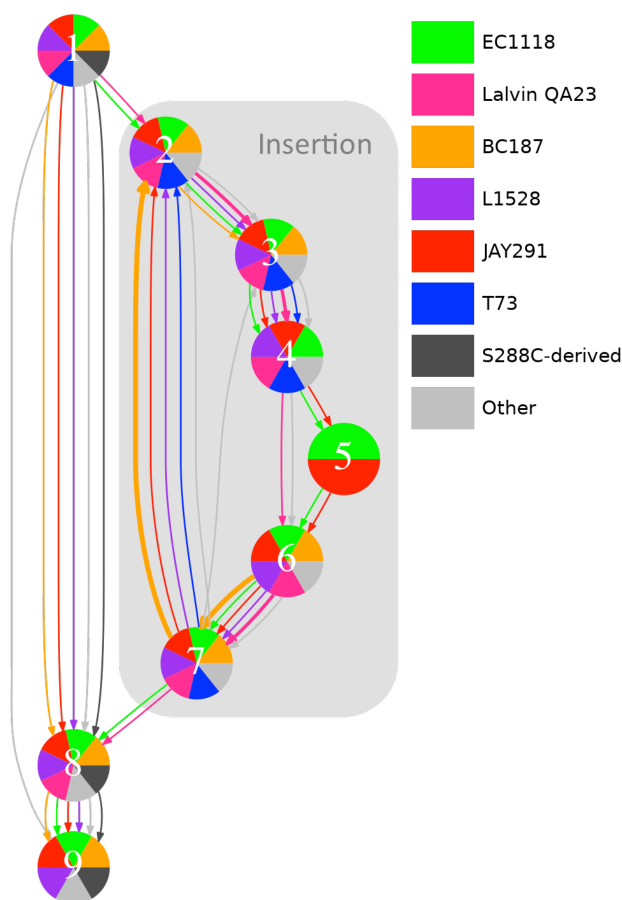

Figure 3.5: The 17kb alcohol-related insertion found in EC1118 on the end of chromosome XIV. Yeast strains are highlighted that share four or more of the FRs (Lalvin QA23, BC187, L1528, JAY291, and T73). All of these strains are involved in alcohol production. The five S288C-derived yeast strains and others are shown as well. Copy number is represented by arrow weight. The inserted region is shaded.

yeast strains used in generating alcoholic beverages and biofuels ($k = 500$, $\alpha = 0.7$, $\kappa = 0$, minsup = 2, minsize = 500, reverse complement included). Of 20 additional alcohol-related yeast strains, 9 shared FRs found in EC1118 in the 17kb insertion region. Five of these strains shared four or more FRs from the EC1118 17kb insertion and are highlighted, along with EC1118, in Figure 3.5.

Several interesting, biological observations are evident from the structure of the

graph. None of the five S288C-derived laboratory lines have the inserted region on chromosome XIV nor in any other region of the genome. While the highlighted strains share many of the 17kb insertion FRs with EC1118, none share the same path. The most closely related path is Lalvin QA23, which not only shares all but one of the inserted FRs with EC1118 but also has these FRs integrated into the same region of the genome. This is evident by the connections between chromosome XIV nodes flanking the insertion (nodes 1 and 8) and the nodes on either end of the insertion (nodes 2 and 7, respectively). The fact that Lalvin QA23 has all of the novel nodes inserted into the same genomic region as EC1118 except one suggests that they shared the same insertion event. While node 5 may have been lost from LalvinQA23, the fact that node 5 is so rare (present only in EC1118 and JAY291 (bioethanol strain) but none of the other 46 yeast strains) suggests that it could represent another novel insertion.

While LalvinQA23 is the only strain with evidence suggesting that it shares an insertion site with EC1118, assembly fragmentation in some lines may prevent us from identifying their insertion site or confirming that their chromosome XIV does not have an insertion. Several lines, however, do show good evidence that the insertion occurred elsewhere. BC187 (wine), L1528 (wine) and JAY291 (biofuels) all have FR paths from node 1 to 8 to 9, just like the S288C-derived lines, indicating that this portion of chromosome XIV is intact and does not contain any insertions. Nevertheless, these strains clearly contain many of the novel FRs required for survival in high alcohol environments.

Indeed, some of these strains show multiple copies of these FRs, which may be important in increasing alcohol tolerance. BC187 has three genomic copies of the node 6 $\rightarrow$ node 7 $\rightarrow$ node 2 path that map to three different scaffolds. Intriguingly, Lalvin QA23, whose insertion so closely mirrors that of EC1118, appears to have

two other genomic regions that contain subsets of the FR path on chromosome XIV. The three Lalvin QA23 regions may be the result of three separate insertion events or may be due to duplication events within the Lalvin QA23 genome followed by deletion or divergence within duplicated regions. The two other regions containing similar sequences in Lalvin QA23 are on chromosome XII (node 2 $\rightarrow$ node 3 $\rightarrow$ node 4) and a scaffold whose chromosome is unknown (node 6 $\rightarrow$ node 7).

Finally, with the exception of the Lalvin QA23, there is rearrangement in the inserted regions compared to EC1118. Intriguingly, the four other strains (BC187, L1528, JAY291, and T73) all have node 7 leading into node 2. These two nodes make up opposite ends of the insertion in EC1118.

Our pan-genomics algorithm allowed us to quickly assay complex insertion events that are important for alcohol tolerance across multiple yeast strains. We were able to quickly identify yeast strains with similar regions inserted and hypothesize about which were independent insertion events. In addition to insertions, we identified complex rearrangements. By adjusting the stringency of the parameters, one could quickly identify which of the inserted regions are most conserved and look for similar but more diverged insertions in other yeast strains involved in alcohol production that did not show a match in this data set. Because of complex rearrangements and independent insertion sites, it would be much more difficult to analyze these alcohol-related insertions using alignments back to a reference strain, especially if the reference strain did not have the insertions.

3.7.2.3 Using FRs for Classification As mentioned in Section 3.6.1, we use SVMs as our supervised learning model. Specifically, we applied two different one-against-rest multi-class SVM classifiers, one with linear kernels and one with degree two polynomial kernels, both with margin parameter $C = 10$. We chose one-against-rest

due to its computational efficiency and interpretability. We normalized the data using L1 normalization and then used stratified cross-validation to evaluate the effectiveness of the model. In this work, we used these SVM configurations to predict the industrial origin for a given yeast strain.

We implemented the SVM models using the scikit-learn Python machine learning library.[3] We then used the models to classify the strains in the yeast data set by their industrial application, or rather *source* from Table 3.2. The data set is composed of 55 strains/examples annotated with nine distinct class labels, one label per example. Given the distribution of the source labels in the data set, we used 2-fold stratified cross-validation, amounting to 10 x 2 stratified cross-validation. For $\alpha \in \{0.7, 0.75, 0.8, 0.85, 0.9\}$, $\kappa = 0$, minsup $\in \{5, 10, 15, 20\}$, and minsize $\in \{5, 10, 25, 50, 100\}$, we computed the micro average Area Under the Receiver Operating Characteristic (AUROC) curve [7] for 10 iterations of stratified cross-validated classifier training using the trapezoidal rule. Additionally, we performed the same experiments while selecting the top 1000, 500, and 250 FRs (features) in the training examples using our multinomial-filter. The box plot of the SVMs' AUROC results is shown in Figure 3.6.

As we can see, both the linear and polynomial kernel classifiers generally have classification power, though the polynomial kernel is consistently better than the linear. We also observe that the multinomial-filter appears to slightly decrease the classification power of the SVMs. An observation not apparent in Figure 3.6 is that the support and size of the FRs seems to directly affect the classification power of the polynomial kernel. Specifically, FRs with low support and large size tend to have much higher classification power across k values. For example, Figure 3.7 shows a box plot of the AUROC results for the polynomial kernel SVM on the $k = 1000$ data for

[3] `http://scikit-learn.org/`

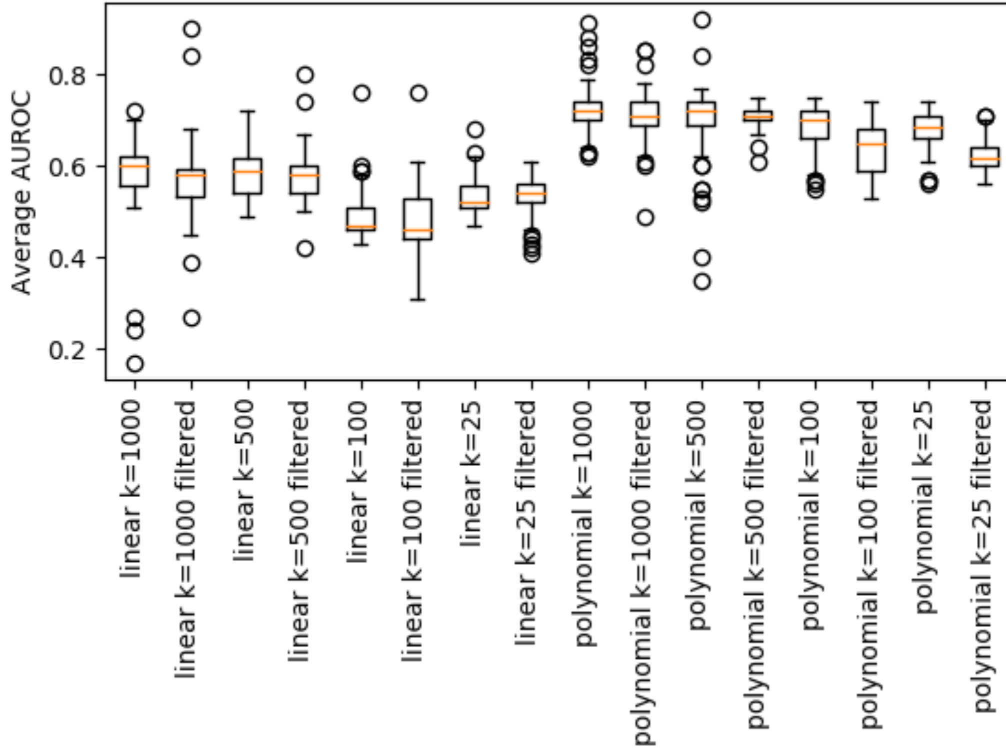

Figure 3.6: Box plot of the AUROC curve results for both SVM configurations (linear versus degree two polynomial kernels) on unfiltered and filtered (top 1000, 500, and 250) *Saccharomyces cerevisiae* iFRs computed with `FindFrs` (using parameters $\alpha \in \{0.7, 0.75, 0.8, 0.85, 0.9\}$, $\kappa = 0$, minsup $\in \{5, 10, 15, 20\}$, and minsize $\in \{5, 10, 25, 50, 100\}$).

only the low support (minsup $\in \{5, 10\}$) and large size (minsize $= 100$) data sets. As we can see, the outliers from Figure 3.6 are in fact these low support, large size data sets. We also observe that, in this case, the multinomial-filter appears to improve the classification power of the polynomial kernel SVM. This suggests that when mining FRs for classification purposes, `FindFRs` should be parameterized to compute iFRs with low support and large size, and the multinomial-filter should be applied.

3.7.2.4 Visualizing Pan-Genomic Space We also applied *multidimensional-scaling* [55], using the `isoMDS` method in R [92], to provide a visual representation of the industrial

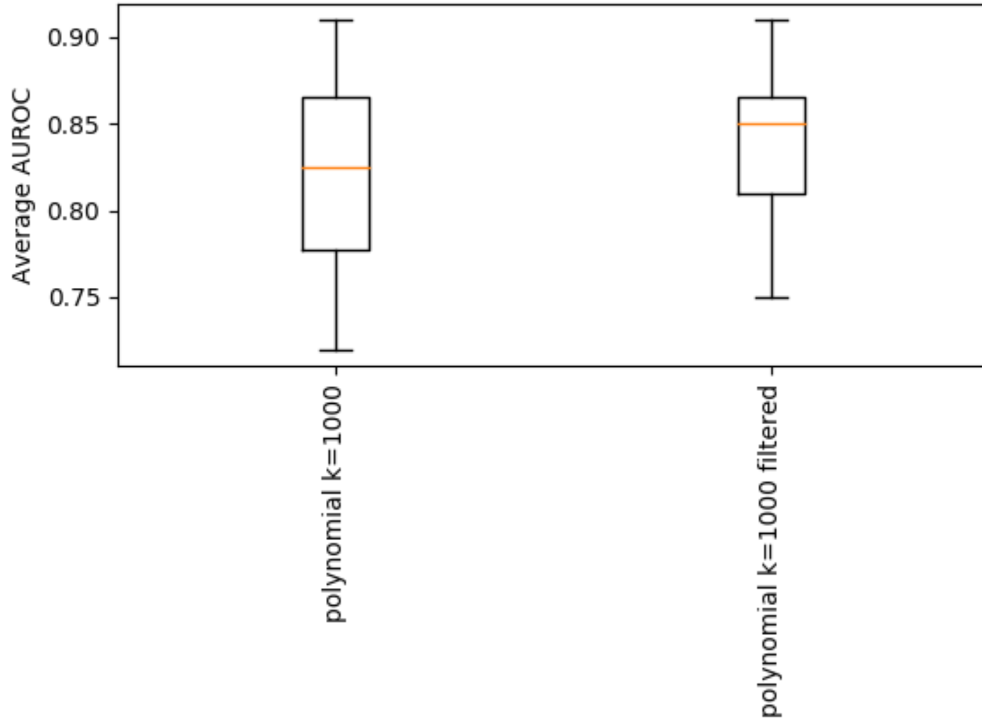

Figure 3.7: Box plot of the AUROC curve results for the polynomial kernel SVM configuration on unfiltered and filtered (top 1000, 500, and 250) *Saccharomyces cerevisiae* iFRs computed with `FindFrs` (using parameters $\alpha \in \{0.7, 0.75, 0.8, 0.85, 0.9\}$, $\kappa = 0$, minsup $\in \{5, 10\}$, and minsize $= 100$).

uses for yeast. For the set of iFRs found using `FindFRs` with parameter values $\alpha = .0.7$ and $\kappa = 0$, the ranking method described in Section 3.6.1 was used to determine the top 500 iFRs by probability (Equation 3.5) for discriminating the nine industrial uses from Table 3.2. Since each industrial usage can be represented as a vector of FR occurrence counts, distances between usages were found using the Canberra method [57, 58] provided by R's `dist` function

$$d(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} \frac{|p_i - q_i|}{|p_i| + |q_i|} \tag{3.6}$$

where $\mathbf{p}$ and $\mathbf{q}$ are industrial usage vectors.

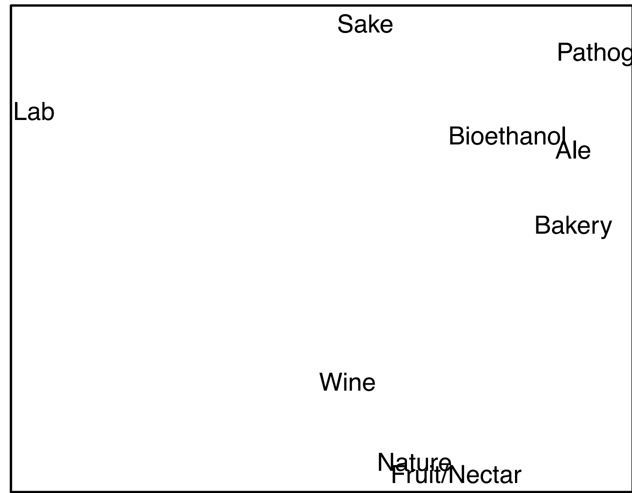

Figure 3.8: Yeast industrial usage multidimensional-scaling plot based on the top 500 discriminative iFRs found with `FindFRs` (using parameters $k = 500$, $\alpha = 0.7$, and $\kappa = 0$).

Figure 3.8 shows the multidimensional-scaling, which provides a visual interpretation of yeast usages based entirely on the aggregate FR content of each group. As can be seen, the laboratory usage is isolated on the left. This is no surprise since these strains have very different selection pressures than the other usages. The alcohols, bakery, and pathogen labels are all pushed to the right, perhaps because these strains have to deal with anearobic conditions. Additionally, they all grow on some grain substrate — barley, rice, corn, wheat — which are themselves closely related [93]. The bottom has wine, fruit/nectar, and nature. Though it is hard to really know what samples the nature accessions were actually growing on, the wine and fruit/nectar certainly have similar substrates.

3.8 Conclusions

Frequented regions provide new approaches to analyze the pan-genomic space. While we have described a few examples showing biological relevance above, there are

many other biological problems to which our pan-genomic method can be applied [17], such as augmenting existing variant calling techniques, aiding in the annotation of new genomes, and better characterizing complex genomic regions within a population.

FRs permit analyzing multiple related assemblies simultaneously without reference bias to find patterns both of divergence (novelty) and conservation. Identification of divergent regions (regions that do not fall into FRs even with lenient parameters) allows the detection of novel genes that are not present in the reference sequence and/or in other assemblies from a species. These genes could represent genes obtained through horizontal transfer, hybridization, or strong positive selection, and may have important adaptive functions. On the other hand, identification of regions that are conserved enables the determination of core gene sets that are required for the species. Determining unannotated regions that are conserved across the species is also important. Such conservation could imply that purifying selection has been active to keep important regions conserved. Such regions could also lead to the identification of new genes or important regulatory elements.

Path-based approaches could also be applied at the amino acid level and potentially at the domain, gene, gene family, operon, or molecule (chromosome or plasmid) level. Our FR approach would be best integrated into a visual tool to help researchers understand and explore pan-genomic data, for example, graph visualizations allowing users to expand iFR nodes into the underlying structure or perform analyses on their genetic content, such as multiple sequence alignment. Furthermore, existing annotation data could be superimposed on the graph to guide the user's inquiry.

CHAPTER FOUR

GENOME CONTEXT VIEWER

In this Chapter, we again consider the problem of mining synteny, but at the genic resolution. We address the synteny mining problem, and subsequently research Questions 2 through 4 from Chapter 1, with an emphasis on Questions 3 and 4, by presenting the Genome Context Viewer (GCV), a visual data-mining tool that allows users to search across multiple providers of genome data for regions with similarly annotated content that may be aligned and visualized at the level of their shared functional elements. By handling ordered sequences of gene family memberships as a unit of search and comparison, the user interface enables quick and intuitive assessment of the degree of gene content divergence and the presence of various types of structural events within syntenic contexts. Insights into functionally significant differences seen at this level of abstraction can then serve to direct the user to more detailed explorations of the underlying data in other interconnected, provider-specific tools.

4.1 Contributions

Expanding on Contribution 3 from Chapter 1, to our knowledge, this is the first tool to use gene family annotations to perform on-demand pan-genomic analyses of genomes distributed across a heterogeneous set of data providers. This approach is known as data federation, and is enabled by the requirement that data providers adopt a common set of gene family annotations and that they provide access to their annotated gene family content through a RESTful API that we have defined. The GCV analyses are performed with pairwise and multiple sequence alignment

algorithms for which we present novel extensions to enable the identification of different types of structural variation, such as tandem duplications and inversions. We also present a novel profile Hidden Markov Model configuration for computing multiple sequence alignments on annotated gene sequences as well as a novel dynamic programming algorithm for computing pairwise synteny blocks between chromosomes represented as annotated gene sequences. We evaluate the GCV by considering two existing deployments — the Legume Information System and the Legume Federation — both of which perform analyses on a variety of legume species distributed across multiple data providers.

4.2 Introduction

In Chapters 2 and 3, we discussed methods for analyzing pan-genomes at the nucleotide level. In this Chapter, we discuss methods for analyzing whole clades, rather than pan-genomes, at the genic level, though these methods may be appropriate for pan-genomes of species that tend to be both large and complex.

The advances in sequencing technology and algorithms that have led to unprecedented amounts of sequenced whole genomes have also led to the widespread availability of annotated whole genome assemblies. These annotated assemblies vary widely in terms of the technologies and algorithms used, the complexity of the genomes targeted, the quality of their representation and the magnitude of the phylogenetic distances among them [9]. For many comparative applications, users of these resources are less concerned with low-level details of sequence alignment than with basic questions surrounding functional content and the genomic contexts in which it occurs. Consequently, there is a need for tools that can facilitate the use of contextualized functional annotation as a unit of search and comparison among sets of genomes spanning diverse taxonomic groups. Furthermore, the genomes within

such a set of data may be curated by different organizations and so may reside in a distributed set of databases. Since the full complement of these data may exceed the storage or computational limits of many users, there is a need for tools that can perform comparative analyses on these data in their distributed state, specifically, by performing heavy computations at each data source while leveraging federated data standards and APIs to enable cross-provider comparative analyses.

Here we present the Genome Context Viewer (GCV), a web-based visual data-mining tool for dynamically identifying syntenic genomic segments and enabling interactive exploration of gene content similarities and differences among distributed collections of annotated genomes.

Using GCV, a simple request using a gene known to reside in a region of interest is sufficient to retrieve all genomic segments with similar gene content (regardless of whether a match is present to the query gene itself), align the genes in the returned segments to account for gene presence/absence, copy number, and structural variation, and present the result in an intuitive interactive view for in depth exploration.

4.3 GCV Application

In this section, we give a high level overview of the GCV application. Design details and algorithms are discussed in detail in Sections 4.5 and 4.6.

A *genome context* is a region considered primarily with respect to the ordering and orientation of its functionally significant elements. The primary visualization of a genome context in GCV is a horizontal track in which triangular glyphs represent genes ordered according to their occurrence in the segment, with directionality indicating orientation and intergenic distances represented by the thickness of connecting lines Colors are assigned to reflect membership in families, providing a

visual overview of homologies within and between tracks. The association of each color with a gene family is presented in an interactive legend that can be used to highlight all members of a family present in the view, or to access more information about a family.

Federation software is software with the ability to aggregate data from disparate sources. GCV uses a service-oriented design to achieve a separation of server-side functions of content match and retrieval from client-side functions of segment alignment and display. This enables federation of data from multiple providers into a single comparative context, depending only on their adoption of a consistent classification of genes into families, or rather, homologous relationships.

The main view of GCV is built to represent a set of genome contexts in terms of their functional content. There are two variations of this view: 1) the *search* view, shown in Figure 4.1, in which a single gene is specified as the focus of a query track and a user-specified number of flanking genes determines the track extent, and 2) the *multi* view, shown in Figures 4.2 and 4.3, built from a user-specified set of genes, each of which serves as the *focus* gene of a genome context, flanked by a user-specified number of genes. In the search view, provider services are invoked to locate segments similar in content to the query. Matched segments are then aligned to the query based on gene family membership and ordering using modified pairwise sequence alignment algorithms. Similarly, in the multi view, the contexts constructed from the user provided genes are clustered based on their gene family content and then all the contexts within each cluster are aligned using a multiple sequence alignment algorithm (Figure 4.2). In both views, the alignment algorithms operate on the gene family alphabet, rather than the underlying sequence, greatly reducing the computational requirements and allowing them to compute alignments on the fly within the responsive user interface. In the case of the search view,

the pairwise alignments are optimal. All parameters for defining acceptable gene content similarity and alignment scoring may be altered by the user to suit their specific application. Additionally, further algorithmic modifications make it possible to correctly align inversions and segmental tandem duplications, events occurring frequently at the scale of multi-gene segments in genomes but which are outside the scope of traditional sequence alignment algorithms [26].

In the search view, pairwise dot plots are used to represent spatial distributions of corresponding annotations and aid in identifying elements lost, gained, or subjected to copy-number alterations. Local plots are composed of the gene family content of the query and result track segments, whereas global plots display all instances of genes from the families of the query track and result track across the chromosome from which the matched syntenic segment was taken. This gives a better sense for the frequency with which members of these families occur outside the matched context and can reveal wider syntenic properties, such as the existence of multiple collinear matches to a region in disparate locations on a single chromosome.

GCV can also display pre-computed and dynamically computed macro-synteny (chromosome scale) blocks. In the search view, these blocks are displayed in a viewer similar to genome browser feature tracks [100, 103], with the region corresponding to the current genome context highlighted. Dragging this to a different region triggers a new search, allowing the user to quickly move to regions that may be of interest for higher-level structural properties, such as breakpoints. Similarly, in the multi view, the GCV can display macro-synteny blocks in a viewer similar to a Circos style plot [56], where data is visualized in a circular layout (Figure 4.3). Here, each chromosome present in the micro view is represents a segment of the circle with arcs connecting syntenic segments between them. Ordering and filtering of macro-synteny tracks is coordinated with the corresponding micro-synteny tracks below.

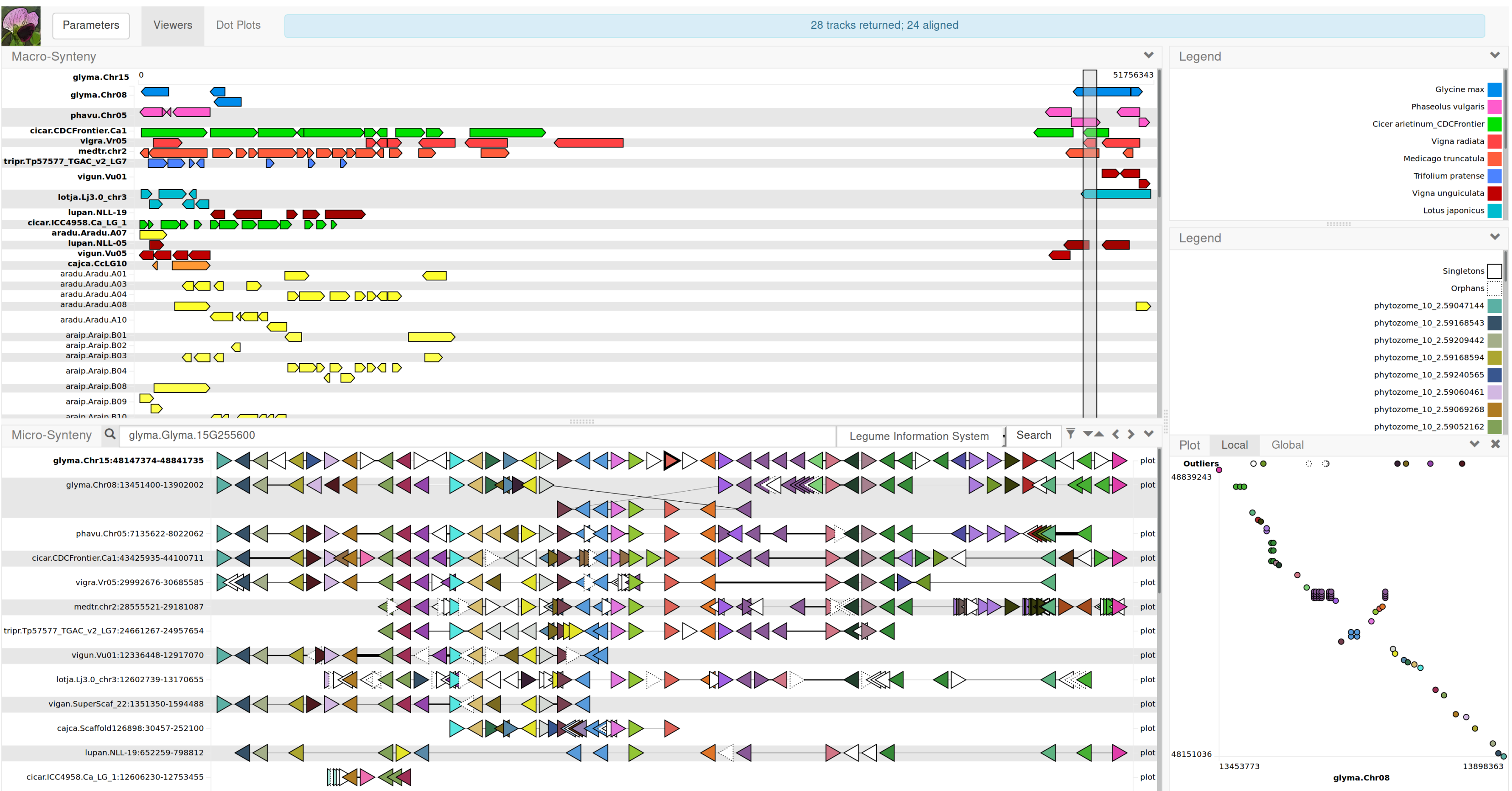

Figure 4.1: A GCV search view exhibiting copy number presence/absence/variation and structural rearrangements. (lower left) Micro-synteny relationships generated from search results aligned with the Repeat algorithm described in Section 4.6.2 to capture inversion. (lower right) A local dot plot of the query track and a selected result track, giving a complementary view of micro-synteny features. (middle right) A legend mapping gene families to colors in the micro-synteny and dot plot visualizations. (upper left) Macro-synteny blocks computed on-demand using the method described in Section 4.6.6. The blocks indicate the chromosome-scale context of the micro-synteny tracks below. (upper right) A legend mapping organisms (genus and species) to colors in the macro-synteny visualization.

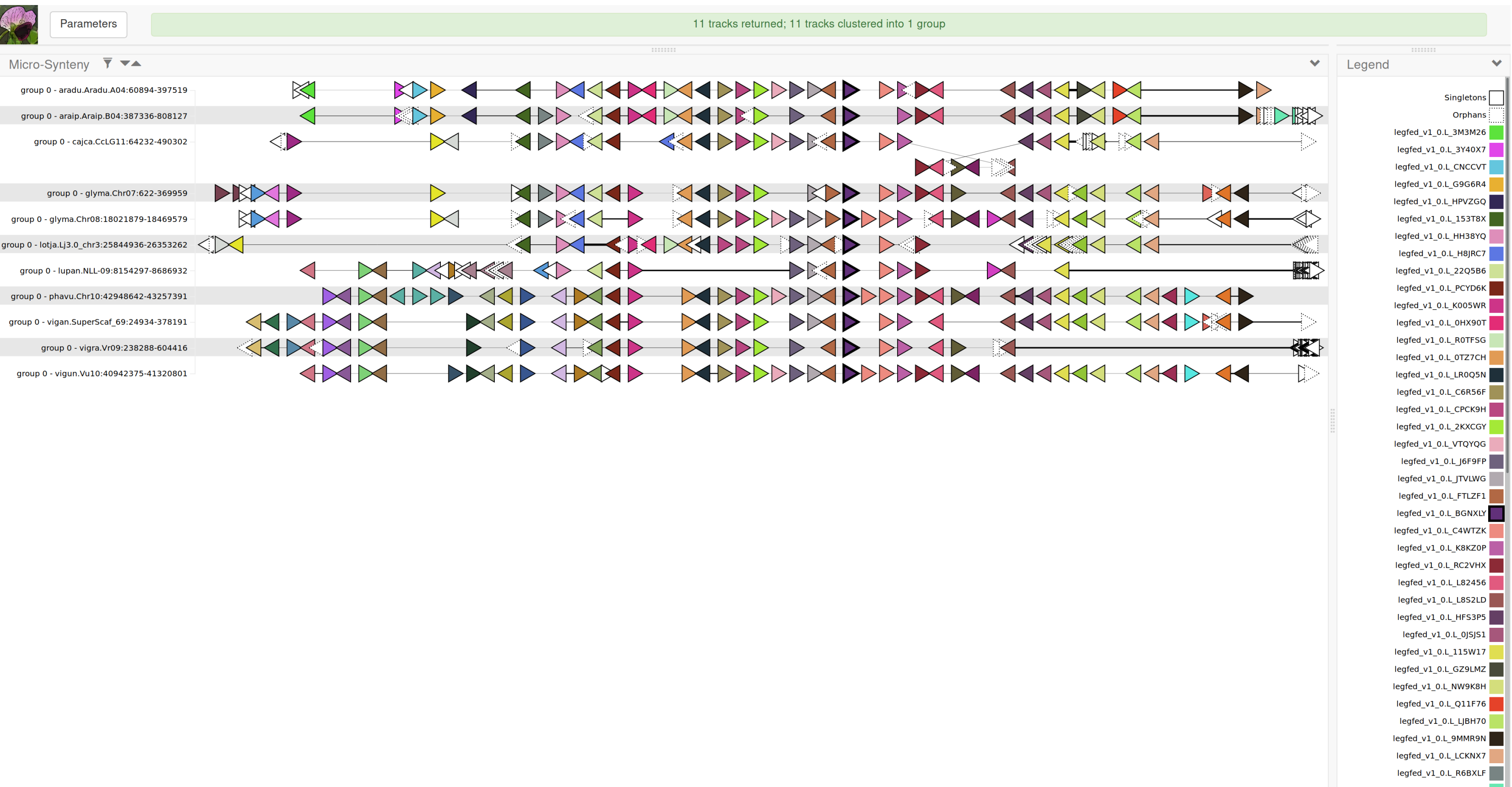

Figure 4.2: A GCV multi view exhibiting copy number presence/absence/variation and structural rearrangements. (left) Micro-synteny relationships generated from user defined tracks aligned with the profile HMM described in Section 4.6.5, which has captured an inversion. (right) A legend mapping gene families to colors in the micro-synteny visualization.

Figure 4.3: A GCV multi view Circos style plot. (left) The Circos style plot for the micro-tracks from multi view shown in Figure 4.2. Each chromosome present in the micro-synteny visualization is represented by a segment of the circle, the extent of which is proportional to the chromosome's physical level (measured in base-pairs). The arcs between segments depict pairwise macro-synteny blocks computed on-demand using the method described in Section 4.6.6. The colors of the blocks depict what chromosome they were computed relative to. (right) A legend mapping organisms (genus and species) to colors in the Circos style visualization.

All GCV visualizations provide a context-menu allowing users to download high-quality images of the visualizations as well as the underlying data for further analysis. Individual elements such as genes, genomic regions, and gene families also provide context-menus whose specific content can be customized to interlink with other resources providing complementary functionality.

4.4 Related Work

The motivation for using gene-families as a unit of search and alignment was two-fold: 1) to emphasize functional content and the genomic contexts in which that content occurs and 2) to make these analyses sufficiently performant to be done on-demand across large taxonomic groups in a federated manner. The merit of the first has been sufficiently discussed elsewhere [66, 68], so we will focus on the second.

The GCV makes use of algorithms to determine segmental similarity and collinearity by use of the pre-estabished gene family assignments. By using pre-estabished gene family assignments, GCV is similar to the visualizations provided in the Eukaryotic Gene Order Browser [66] and the gene family pages of Phytozome [32]. By using algorithms to determine segmental similarity and collinearity, GCV is comparable to tools that are concerned with problems of whole genome synteny comparison, not only because the macro-synteny tracks can be computed dynamically, but also because of the extent of the micro-synteny tracks which span large segments of the target chromosomes' DNA.

There exist a variety of web-based tools for pairwise and multiple genome synteny search and comparison. The three that are perhaps most related to the GCV are the Plant Genome Duplication Database (PGDD) [59], CoGe's GEvo [69], and Genomicus' PhyloView [68].

PGDD is a database characterizing whole genome duplication events in plant

genomes, providing both intra- and inter-genome syntenic relationships. Similar to the GCV, users can perform synteny searches by providing a query gene or can generate a dot plot by selecting two genomes to compare. A locus search[1] will yield a set of precomputed matches between anchor genes for a genomic window of specified size centered on the query gene, rather than an explicitly aligned representation of the query to the resulting tracks (Figure 4.4). PGDD uses the MCScanX algorithm [109] to precompute collinear blocks, where candidate anchors are based on either pairwise BLAST [2] of the gene models or a clustering of genes into homologous groups. In Section 4.4 we validate the use of our gene family assignments as a surrogate for direct pairwise comparison by computing whole chromosome synteny blocks at the genic resolution. We found that these synteny blocks are similar to those produced by the technique used by PGDD, suggesting that the macro/micro-syntenic blocks reported in the GCV are valid and that precomputed sequence-level / whole genome blocks are not necessary for deriving blocks within a certain locale.

CoGe's GEvo (Genome Evolution Analysis) is one of a large suite of tools for genome synteny analysis.[2] GEvo is the tool within CoGe that is probably most comparable to GCV, as users can perform a search by specifying some genomic feature and flanking region as a search query and selecting what alignment algorithm(s) and corresponding parameters to use. Results are then displayed as pairwise mappings of matched blocks between sequences. The query and search sequences can be loaded from CoGe's database, NCBI, or provided by the user; similarly, CoGe allows any user to upload any number of genomes to use as the target to their analyses, which allows it to be used regardless of the specific taxonomic focus of any given user. A GEvo search can also be initiated from a pair of genes previously determined to reside in a syntenic

[1] `http://chibba.agtec.uga.edu/duplication/index/locus`
[2] `http://genomeevolution.org`

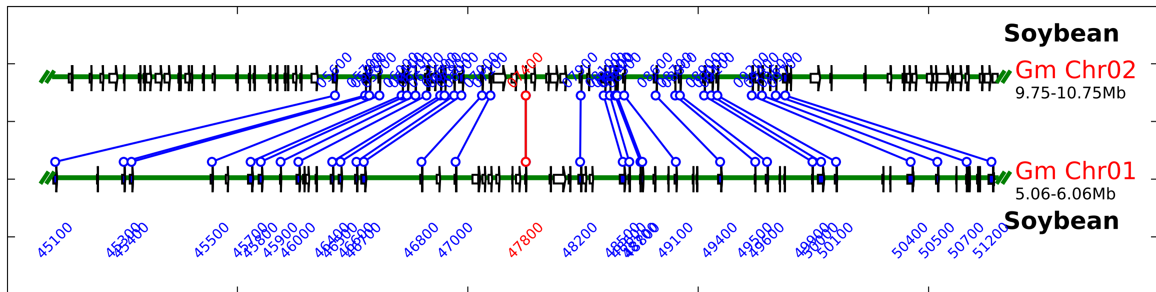

Figure 4.4: The soybean chromosomes 1 and 2 comparison from region showing the same inverted segment in Figure 4.7, produced using the PGDD locus search. Noteworthy is that the inverted block displayed in GCV using its repeat and track merging algorithms shows up as an unmatched set of anchors (with the exception of the central gene in the inverted segment). This has the result that the choice of any of the genes within the interval as the locus to be searched fails to find the two blocks in which they are embedded as being syntenic. GCV's approach finds the blocks as candidates due simply to their similar gene content and initially ignores the ordering which has been disrupted by the structural variation, then applies a client-side modified alignment algorithm to detect the presence of the inversion and display accordingly.

region by one of the other tools in the CoGe suite, for example, from a dot in the whole genome comparison dot plots produced by CoGe's SynMap tool, which makes it convenient as a mechanism for generating sequence-level similarity in restricted regions that are predetermined to contain syntenic content. The Gobe viewer for GEvo output is implemented in Flash, and unlike the gene family strategy taken by GCV, emphasizes comparisons of high scoring pairs between pairs of sequences and their relationship to gene structural elements — such as exon structure — making the view more detailed than GCV and hence a little more complex to interpret when large numbers of comparisons are in play. This could obscure the identification of structural events, such as copy number variation and presence/absence variation among gene clusters; although, it makes it possible to see sequence-level events below the level of those that impact annotation, which would be invisible to GCV. The ability to

perform on-demand sequence comparisons for genomic segments of interest provides a naturally complementary functionality to the approach taken by GCV and we are developing approaches for utilizing CoGe web services as optional plugins and linkouts through the service framework used by GCV.

Genomicus' PhyloView is similar to GCV in that it compares sequences of genes based on their gene family content. In fact, the alignment view is built from a tree representing the phylogenetic relationships among the returned tracks, where each track is displayed alongside the node it represents in the tree. This is an excellent feature and well suited to the ancestral reconstruction of genomic segments – a key strength of the system. However, for the purposes of federating data across providers, a tight coupling of tracks even with a predetermined tree makes integration with multiple data sources non-trivial. Additionally, rather than compressing, misaligning, or inverting inexact matches, Genomicus simply omits content from other tracks not matching a gene family in the query. This could prevent the user from identifying interesting structure present in many tracks other than the query.

The feature that most fundamentally distinguishes the GCV from these tools is that it only requires each genome's annotations have their gene family assignments pre-computed, which can be done independently on each genome, assuming that the family definitions are sufficiently broad to capture most gene content of interest within the taxonomic group. This allows GCV to easily support data federation, only requiring that all service providers have come to agreement on a common set of gene family definitions. This model enables the user to recognize events like copy-number variation and presence/absence variation across large sets of genomic segments from taxonomic groups that span multiple genome data providers.

4.5 Architecture

GCV is a client-side single page Web application that can be run locally or be integrated into a website. It consumes data from one or more service providers that implement the interface defined in a RESTful API.[3] Once it has aggregated data from the providers, GCV creates visualization-specific data representations (micro-synteny alignments, dot plots, and macro-synteny tracks), filters these data according to user-defined criteria, and visualizes the results. The software architecture and data flow are depicted in Figure 4.5.

4.5.1 Technology Stack

GCV was implemented using modern Web standards technologies, specifically, Angular,[4] ngrx/store,[5] and D3.[6] Angular is used to retrieve data from service providers, manage UI components, and mediate communication between the visualizations and the rest of the application. ngrx/store is used to manage application state and give GCV explicit data flow, and D3 is used to draw the various visualizations. Furthermore, the visualizations and their inter-visualization interaction mechanisms have been encapsulated in their own JavaScript library so they can be used independently of GCV.

GCV is service implementation agnostic, only requiring that the services it uses adhere to the RESTful API. Even so, the service implementation used by LIS and LegFed (see Section 4.7) is freely available so that users with similar infrastructure need not re-implement GCV services. The services are implemented as a Django

[3]See `https://github.com/legumeinfo/lis_context_viewer` for the full API.
[4]`https://angular.io/`
[5]`https://github.com/ngrx/store`
[6]`https://d3js.org/`

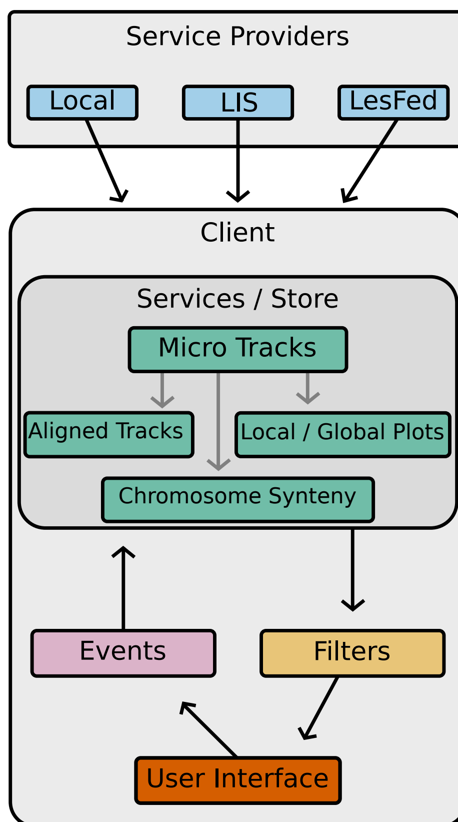

Figure 4.5: The software architecture and data flow of the GCV. GCV (an Angular application) consumes data from one or more service providers, one of which may be the user's own computer (local). Angular services are responsible for requesting data from service providers, aggregating the results, and storing them in ngrx/store. For example, in the search view, whenever new micro track data (blue) is acquired, visualization specific representations are made (green). These representations are then filtered by user controlled criteria and consumed by Angular components (the UI) which draw the data with D3. User interaction with the UI can trigger certain events that will notify the services to update the representations or acquire new data, such as changing alignment parameters or search parameters, respectively. Figure created by Brittany Fasy.

application[7] on top of a Chado database [79].

[7] https://www.djangoproject.com/

4.5.2 Services

The providers that GCV uses must implement one or more of the RESTful API services. Here we briefly describe the core set of services.

- *multi micro-synteny tracks*: Takes a set of focus genes and the number of neighbors that should flank each gene. Returns the set of tracks centered about the focus genes.

- *search micro-synteny tracks*: Takes a query track, that is, an ordered list of gene families and search parameters: minimum number of matched families and maximum number of non-matched families between any two matched families. Returns all micro-synteny contexts in the database that meet the given parameter constraints.

- *gene to query*: Takes a focus gene and the number of neighbors as input. Returns the track centered about the given focus gene.

- *macro-synteny tracks (static)*: Takes a reference chromosome identifier and optionally a set of aligned chromosome identifiers as input. Returns the synteny blocks of the aligned chromosomes with positions relative to the reference chromosome. If no aligned chromosomes are provided, it returns all synteny blocks relative to the reference chromosome in the database.

- *macro-synteny tracks (dynamic)*: Takes a reference chromosome as an ordered list of gene families, and search parameters: minimum number of matched families, maximum number of non-matched families between any two matched families, and the maximum number number of members a family may have before it is filtered (see Section 4.6.6). Optionally, the user may provide a set of aligned chromosome identifiers to limit the output to. Returns the synteny

blocks of the aligned chromosomes with positions relative to the positions of the families in the provided reference chromosome.

- *global plot*: Takes a set of gene families and a chromosome identifier as input. Returns all the genes on the specified chromosome that are members of the given gene families.

An important caveat to consider is that the data provided by a service provider may actually be aggregated from multiple curators or data-stores, as is the case with LIS and LegFed. To enable better interoperability with other sites, such as the primary repository of a particular genome, GCV can use services to load links to relevant external sites. For example:

- *gene links*: Takes a gene identifier as input and returns a list of external links that the gene can be passed to for further inquiry.

4.6 Algorithms

A variety of existing and novel algorithms are used both in GCV and the provided services implementation. Here we present seven such algorithms that are integral to the search and multi views and illustrate the data flow depicted in Figure 4.5.

4.6.1 Track Search via AFS Supporting Paths

Although the implementation of GCV services is left to the service providers, due to the search view's central role in GCV and the non-trivial nature of the corresponding service, we discuss how the open-source example server implements the micro-synteny search service.

The problem of finding micro-synteny tracks with similar gene family content and ordering to the query track is an instance of the Fixed-Radius Near Neighbors

problem [6]. In the example implementation, the radius is defined by the query parameters — minimum number of *matched* gene families and maximum number of non-matched, or *intermediate*, families between any two matched families — so the search space is 2-dimensional. Gene family ordering (edit distance) could be a third dimension, but we leave the consideration of ordering beyond computing the number of genes between matches as a task for the front-end. This is because track similarity in this sense is dictated by the choice of alignment algorithm and corresponding parameter values. Thus, the tracks returned by the micro-synteny search service as we have defined it are invariant with respect to choices of alignment algorithm and parameters, so changing these in the client does not require a new set of server requests.

It can be seen that the problem of finding micro-synteny tracks as we have defined it is a variation of the AFS supporting paths problem from Chapter 2. Here, though, the ϵ_r constraint is a minimum number of nodes that each supporting path must traverse (*matched*), rather than a fraction, and there is no ϵ_c constraint on the number of subpaths that must support each node in the anchor path. The *intermediate* constraint is equivalent to the AFS μ constraint and minsup $= 1$. As we will see, this variation of the AFS supporting path problem is tractable.

The example implementation uses an exact algorithm that works as follows: Given a micro-synteny search query (anchor path), specifically, the list of gene families present in the query track, the algorithm iterates the ordered list of gene families of each chromosome in the database. When a gene family that matches one of the query families is found, a new candidate track is created. The candidate is grown by iteratively adding the next family in the list to the track until the number of families added since the last match is greater than *intermediate*. The candidate track is then trimmed back to the last matched family and returned if the number of matched

families it contains is greater than or equal to *matched*. The algorithm then continues iterating the ordered list of gene families at the family after the last family iterated by the previous candidate growth.

Since each chromosome in the database may have several thousand genes, the example implementation is optimized as follows: First, the algorithm leverages the indexing mechanisms of the underlying database to efficiently find all instances of the query gene families in the ordered list of gene families for each chromosome and memoizes each matched family's position in its corresponding list. Then, the iterative algorithm is applied to each chromosome's ordered list of matched gene families if the list contains at least *matched* number of families. Candidate tracks are grown as before, but the memoized position data is used to compute how many non-matched families lie between the last and next matched families, rather than iterating the non-matched families. A batch query is then performed to fetch all the non-matched families for the candidates that satisfy the *matched* parameter.

4.6.1.1 Time Complexity In the worst case of the optimized version of the algorithm, the entire ordered list of gene families of each chromosome in the database must be iterated, taking $O(L)$ time, where L is the total length (number of genes) of all the chromosomes in the database. The time complexity of the gene family search and retrieval step will vary between database technologies.

4.6.2 Merging Alignments

In the search view, once a set of tracks with similar gene family content to the query track has been retrieved, each track is aligned to the query to emphasize the syntenic relationships and structural differences between the tracks. These alignments can be computed with either the Smith-Waterman or the Repeat local alignment algorithms. The Repeat algorithm [26] is an extension of the Smith-Waterman

algorithm [102]. Specifically, rather than finding the highest scoring local alignment, it finds all local alignments whose scores are above a certain *threshold*. In this work, we extended the Repeat algorithm to identify inversions whose reversal in the containing sequence improves the sequence's alignment score.

In essence, the extension works by first aligning both the forward and reverse orientations of a sequence to the reference with the Repeat algorithm. All resulting forward alignments are then compared with all reverse alignments for shared gene content, which can be done efficiently with an interval tree [18]. When a reverse alignment is found to have shared gene content with a forward alignment, memoized suffix scores from the alignments' traceback matrices are used to determine if replacing the inverted sequence in the forward alignment with the reverse alignment will improve the forward alignment's score, or vice versa. If so, the forward and reverse alignments are *merged* by replacing the inverted sequence and updating the memoized suffix scores and alignment score.

4.6.2.1 Time Complexity Let n be the length of the sequences being aligned, F be the set of forward aligned segments, and R be the set of reverse aligned segments. Since aligned segments can only overlap in one axis of the scoring matrix used by the Repeat dynamic program [26], the number and length of the aligned segments in the overlap axis cannot exceed the length of the sequence in the other axis. Since both sequences are length n, we observe that $|F|, |R| < n$ and $\sum_{f \in F} |f|, \sum_{r \in R} |r| \leq n$. We construct an interval tree for F and then search for overlaps with each interval in R, requiring $O(|F| \log |F|)$ and $O(|R| \log |F| + |R| m)$ time, respectively, where m is the numbers of intervals that overlap with any one interval in R, so $m \leq |F|$ but we expect it to be at most 1. Lastly, computing the score of an inversion and performing the inversion takes linear time in the length of the overlapping segments from R.

In the worst case, it takes $O(n)$ time to perform all inversions. Since $|F|, |R| < n$, the $O(n^2)$ complexity of the Repeat algorithm still dominates the run-time.

4.6.3 Alignment Coordinates

An extension that we have applied to both the Repeat and Smith-Waterman alignment algorithms is the assignment of coordinates to the gene sequences based on their alignments. It works by *positioning* all resulting alignments relative to the reference sequence. Specifically, a matched character is given the position of the character it matched in the reference sequence and inserted characters are given a position between that of the reference characters they were inserted between. For example, given gene family reference $\tau\alpha\kappa\gamma\gamma$ and sequence $\alpha\gamma\kappa\kappa\kappa\gamma$,[8] the following hypothetical forward alignment would be positioned as:

	position in reference:	0	1	2	3				4
alignment	reference:	τ	α	κ	γ	-	-	-	γ
	sequence:		α	-	γ	κ	κ	κ	γ
	position in reference:		1		3	3.25	3.5	3.75	4

The sequence's forward alignment positioning indicates that it spans the character interval 1-4 in the reference.

By positioning all alignments relative to the reference sequence, they are normalized to the same coordinate space. These normalized coordinates are used to position the tracks in the micro-synteny search visualization.

4.6.3.1 Time Complexity

Let n be the length of the sequences being aligned. The alignment coordinates could be computed during the alignment dynamic program by filling a coordinate matrix in conjunction with the score matrix. This would

[8] Greek characters are used only for example purposes. In practice the gene families are matched on their unique identifiers.

require $O(n)$ additional space and $O(1)$ additional time if using the space efficient divide and conquer variations of the alignment algorithms [81]. Alternatively, the alignment coordinates could be computed by iterating the final alignment. The longest alignment possible has length $2n - 1$, meaning computing coordinates would require $O(n)$ additional time. Either way, the $O(n^2)$ complexity of the Smith-Waterman / Repeat alignment algorithm still dominates the run-time.

4.6.4 Clustering Tracks

The multi view displays genomic contexts for a given set of genes. Though the gene set can be arbitrary, it can also contain homologous genes, in which case there will likely be multiple syntenic genomic contexts present in the view. Since this is the intended use of the multi view, we group together contexts in the client that are likely syntenic to make it easier for the user to decipher the evolutionary relationships among the tracks.

To compute the groupings, we use the Frequented Regions algorithm from Chapter 3. In this case, we construct a graph from the gene family content of the tracks, where each family is represented by a unique node and an edge exists between nodes if they are sequential in one or more of the tracks. Unlike the nodes of a de Bruijn graph, a gene family is much more likely to participate in more than one FR. Since the FR algorithm is agglomerative, we accommodate this nuance by repeatedly applying the algorithm. Specifically, after constructing the graph, we use Algorithm 3.3 from Chapter 3 to construct a hierarchical clustering of FRs. We then perform a breadth-first search similar to Algorithm 3.4 from Chapter 3 to identify the largest (most graph nodes) interesting FR in the hierarchy. This will be a root FR, that is, an FR that is not a sub-region of another, larger interesting FR. Once the largest interesting FR has been identified, the search continues down the hierarchy

rooted at the identified FR and searches for interesting sub-FRs that are supported by tracks that do not support the root. This forms a hierarchy of supporting tracks that are homologous within some subset of the root FR's gene families. All such supporting tracks are grouped together and removed from the graph. The FR algorithm is then applied to the updated graph, and so on, until no valid FRs are identified or the graph has been reduced to the null graph.

By using this approach, a gene family that has already participated in an interesting FR may remain in the graph if one or more tracks that contain the family have yet to support an interesting FR, effectively making this approach a *fuzzy* variation of the FR algorithm [25]. Here, the degree of a gene family's membership in an FR is defined as

$$\text{md}(\text{FR}_j, f) = \frac{\text{support}(\text{FR}_j, f)}{\sum_{\text{FR}_i \in \mathbf{FR}} \text{support}(\text{FR}_i, f)} \tag{4.1}$$

where md is a membership degree function that returns a value between 0 and 1, FR_j is an FR, f is a gene family, support is a function that computes the number of supporting paths in an FR that support a gene family, and **FR** is the set of all FRs returned by `FindFRs`.

4.6.4.1 Time Complexity Since the minimum number of supporting paths an interesting FR may have is 2, in the worst case the FR algorithm will be run $\frac{|P|}{2}$ times, where P is the set of track paths. Using the basic version of the FR algorithm (no graph matching or parallelization), this gives a run-time complexity of $O(|P|LV + |P|V^2 \lg V)$, where L is the total length of the paths in P and V is the total number of graph vertices.

4.6.5 Multiple Alignment of Tracks

Similar to the search view, the multi view emphasizes the syntenic relationships and structural differences between the tracks within a group via track alignment. In this case, there is no query track, or *reference*, so a multiple alignment of the tracks is performed. Although there exist a variety of methods for multiple sequence alignment, such as progressive, iterative, and consensus based methods [35, 39, 83], we opted to take a Hidden Markov Model (HMM) approach due to computational speed [77] and to gain finer control over the alignment for visualization purposes.

Our approach is similar to that described in [26]. Specifically, the multiple sequence alignment is computed by iteratively training a profile HMM. The topology of the model is similar to that of [54], where, in addition to begin and end states, there is a visible match state and corresponding hidden delete state for each column in the alignment, and a hidden insertion state between each match-delete state pair, as depicted in Figure 4.6.

The following model configuration and training procedure were empirically determined to be effective for our specific application: Given the list of tracks to align, the initial HMM topology is derived from the first track in the list, or rather, a match-delete state pair is added for each gene in the track, in addition to flanking insertion states. Each match state M's emission probabilities are initialized as follows:

$$\forall \sigma \in \Sigma, e_M(\sigma) = \frac{c_M(\sigma)^{1.5}}{\sum_{\sigma \in \Sigma} c_M(\sigma)^{1.5}} \tag{4.2}$$

where Σ is the alphabet of all gene families present in the list of tracks, e_M is the emission probability function for match state M, and c_M is a function that emits a count of the number of times σ has been matched to state M during training. Note, c_M is initialized with a pseudo-count of 1 for $\forall \sigma \in \Sigma$, which means before

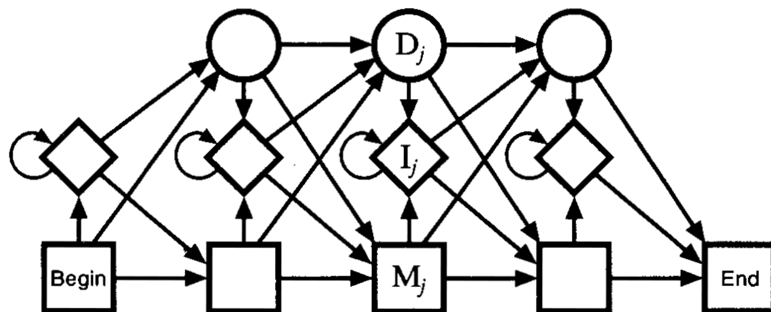

Figure 4.6: The topology of the profile HMM used for multiple sequence alignment of GCV. The square nodes in the bottom row depict the match states. For example, the node with the label M_j is the jth match state. The diamond nodes in the middle row depict the hidden insertion states, and the circular nodes in the top row depict the hidden deletion states. Similar to the labeled match state, the nodes labeled I_j and D_j depict the jth insertion and deletion states, respectively. With the exception of the begin and end states, each state $*_j$ transitions to states M_{j+1}, I_j, and D_{j+1}, where $*$ denotes a match, delete, or insertion state. This means insertion states transition to themselves, enabling support for alignments containing arbitrarily long insertions. Image from [26]

training $\sum_{\sigma \in \Sigma} c_M(\sigma)^{1.5} = |\Sigma|$. It was found empirically that using an exponent of 1.5 caused match state emission functions to rapidly converge to a coherent set of gene family emission probabilities without being overly exclusive.

The transition probabilities out of each state are initialized so that any observation has equal probability of transitioning to each of the neighboring states, accounting for the observations' initial emission probability from the match state ($\frac{1}{|\Sigma|}$). Excluding the last match, delete, and insertion states, each state S has transition probabilities:

$$t_{S,D} = t_{S,I} = \frac{1}{2} - \frac{|\Sigma|}{2(1 + |\Sigma|)} \tag{4.3}$$

$$t_{S,M} = \frac{|\Sigma|}{1 + |\Sigma|} \tag{4.4}$$

where $t_{S,*}$ is the probability of transitioning from state S to some other state $*$ and D, I, and M denote delete, insertion, and match states, respectively. The transition probabilities from the last match, delete, and insertion states to the end state are similarly derived. Since this configuration initially yields equal probabilities of transitioning from one state to another for any arbitrary observation and state, ties between state transition probabilities are resolved by the absolute ordering of the state types:

$$M \prec I \prec D. \tag{4.5}$$

Once the model is configured, it is iteratively trained by aligning each track from the list of tracks to the model. The forward and reverse orientations of each track are aligned to the model using the Viterbi algorithm [107], which returns the most probable path through the model, or *Viterbi path*, that could have generated the given track. Whichever orientation has a higher probability is kept. In the case of a tie, the forward orientation is preferred. We note that since the model contains loops (Figure 4.6), the Viterbi algorithm is "loopy", meaning the inferred alignments are approximate solutions [80].

After each alignment the model is updated as follows: First, insertion states that are traversed by the Viterbi path are converted into match states, one for each gene family that traversed the insertion state and is known to occur in more than one track. The corresponding deletion and flanking insertion states are added as well. The Viterbi path now traverses these match states, rather than insertion states. Next, the count functions of the match states traversed by the Viterbi path are updated, including that of the newly added match states.

Once the model is trained, the final multiple alignment is generated by aligning

each of the training tracks to the final model using the Viterbi algorithm. Again, both the forward and reverse orientations of the tracks are aligned. This is because we want to identify inversions in the tracks, if any, as we did with the Repeat algorithm in Section 4.6.2. Here we identify inversions by merging the forward and reverse orientations' Viterbi paths. This is done by traversing the track and using its alignments to generate a corresponding list of f's, r's, and t's – an f if a gene's forward match state emission probability is greater than its reverse match state emission probability, an r if its reverse match state emission probability is greater than its forward match state emission probability, and a t otherwise, for example, if the probabilities are tied or the element does not traverse a match state in either orientation's Viterbi path. Islands — a single character or chain of t's flanked by two of a different character, such as rfr or $ftttf$ — are then flipped to the opposite character. In the case of ambiguity, f is preferred. The final *orientation list* is then traversed and a merged Viterbi path is generated. Note, if an inversion is present then the generated Viterbi path will not be a valid path through the profile HMM. This is fine since the only remaining function of the Viterbi path is to assign alignment coordinates to the elements of the track, as was done for the pairwise alignments in Section 4.6.3. In this case, character positions are derived from the positions of the match states in the profile HMM, rather than from a reference sequence.

The quality of the alignment produced can be affected by the order in which the tracks are aligned during training. For this reason, a *guide tree*, in which the relationship among the sequences are represented, is used to dictate the ordering [77]. Typically the guide tree is determined using a cheap clustering algorithm, such as neighbor joining, but here we use the group's FR hierarchy from Section 4.6.4 as the guide tree. The ordering of the tracks is derived from the first interesting FR in the hierarchy each track supports and by the number of nodes in that FR they support.

We acknowledge that this method of training the profile HMM completely over-fits the model to the training data. Since the sole purpose of the model is the optimal visualization of the genomic contexts used to train the model, over-fitting the model to the training data is the goal.

4.6.5.1 Time Complexity The initial profile HMM is constructed in $O(|l_0||\Sigma|)$ time, where l_0 is the track from which the model topology is derived. Although the Viterbi algorithm is a dynamic program, it can be efficiently implemented as a message passing algorithm over the topology of the profile HMM. Here, specifically, each node passes one message to each of its three out-neighbors and each node must process one message from each of its three in-neighbors. The nodes incident to the start and end nodes pass/process fewer messages. Since each track's Viterbi path could possibly traverse only insertion and deletion states, causing the model to grow by the length of the track at each training iteration, this gives a worst case running time of $O(|L|\sum_{l \in L}|l|)$, where L is the list of tracks being aligned. This is highly improbable since the tracks being aligned are syntenic, as determined by the FR algorithm, so there must be some level of preservation of gene family content and ordering among them. Generally, we expect the running time of the Viterbi algorithm to be $O(|FR|)$, where $|FR|$ is the size of the interesting FR (number of nodes) that corresponds to L. Lastly, each track's Viterbi paths must be iterated during training and during the final alignment. In the aforementioned worst case, this will take a total of $O(|L|\sum_{l \in L}|l|)$ time, but we expect this to take $O(|L||FR|)$ time.

4.6.6 Dynamic Chromosome-Scale Synteny Blocks

Though the GCV can consume synteny blocks that are pre-computed at the DNA resolution, each service provider may only provide such blocks for chromosomes whose DNA resides in their data store. This means to make blocks available between all

pairs of genomes across data stores, each data store must have copies of all other data stores' genomes or these blocks must be computed on-demand. The first approach would impractically replicate data across all the data stores while undermining the federated data model of the GCV, and the second approach is computationally infeasible at the DNA resolution.

In Section 4.4, we show that annotated gene content may serve as a surrogate for DNA content when computing chromosome-scale synteny blocks between pairs of related species. This is done with the MCScanX tool [109]. Though MCScanX demonstrates the efficacy of such an approach to computing chromosome-scale synteny blocks, it still requires data computed at the DNA resolution as input and so is not practical for the federated, on-demand computational model of the GCV. Here, we present a dynamic program that computes synteny blocks between pairs of chromosomes represented as sequences of gene families.

Given two sequences of gene families, the algorithm begins by iterating one of the sequences and making a data structure that maps each gene family to the set of positions in the sequence at which the family occurs. Then, the other sequence is iterated. For each family iterated, if the family exists in the map data structure, then a set of points is made with the family's position in the sequence being iterated and each of the positions it maps to in the data structure. For example, given gene family sequences $\tau\alpha\kappa\gamma\gamma$ and $\alpha\gamma\kappa\kappa\kappa\gamma$, the following points would be generated:

$$
\begin{array}{ll}
\tau & \\
\alpha & (1,0) \\
\kappa & (2,2),\ (2,3),\ (2,4) \\
\gamma & (3,1),\ (3,5),\ (4,1),\ (4,5)
\end{array}
$$

We note that the user may specify a family size limit, or *mask*, which prevents points from being made for families that have more than the specified number of members

in either of the sequences. This is to prevent overly broad families from obfuscating otherwise coherent synteny blocks. For example, some data providers may assign the same dummy family to all genes that do not belong to a family in order to preserve the positional distances between families in the chromosomes' gene family sequences.

Next, we want to construct chains of points whose x and y coordinates are within a user specified distance of each other and that are monotonically increasing. Each such chain represents a common subsequence of families in the given sequences, or rather a synteny block, so we want to maximize the chain's length. This is done by constructing a directed acyclic graph, specifically a tree, via dynamic programming. The dynamic program uses two lookup tables, one that maps each point to its parent point in the tree and one that maps a point to the length of the longest path that ends at it. The tree is constructed by iterating the points in increasing x, y order. For each point iterated, all the points it dominates whose x and y coordinates are within the user specified distance are considered candidate parent points. The candidate with the longest path length becomes the point's parent and the point's path length becomes the candidate's path length $+ 1$. In the case where the longest path length is shared by two or more candidates, the slope of the line formed by each candidate and the point is used to resolve the tie. Specifically, the candidate whose line slope is closest to 1 is selected. We do this in hopes of preventing parallel chains from interfering with one another, for example, in the case of repeated sequences, such as tandem duplications. If a point has no candidate parents, then it is its own parent and is given a path length of 1.

The final chains are identified from the tree by iterating the points in longest-path-first order. For each point iterated, if the point has not already been identified as part of a chain, then the chain of parents starting at that point is followed until a point is encountered that is already part of a chain or that is its own parent. If the

chain is at least as long as the minimum length specified by the user, then each point in the chain is marked as being part of a chain and the chain is returned.

4.6.6.1 Time Complexity Given sequences of length m and n, in the worst case $m * n$ points will be generated in $O(mn)$ time. Not only is this extremely unlikely, it is also biologically uninteresting since both sequences would be composed of the same single gene family. To avoid such scenarios, the user would provide a value for the *mask* parameter. Then the worst case is that each gene family occurs mask number of times in both sequences, meaning $mask * \min(m, n)$ points will be generated in $O(mask * \min(m, n) + |m - n|)$ time. In practice, we expect the set of points to be sparse relative to the theoretical upper bound, even without masking. From here, we denote the total number of points generated as P.

The chaining construction algorithm described can be efficiently implemented using a (sparse) binary matrix, where a cell has a 1 if the point corresponding to its coordinates was generated, and a 0 otherwise. As points are generated, they are added to the matrix, taking $O(P)$ time. In the worst case, each point dominates $dist^2$ candidate points, where *dist* is the user specified maximum distance between chained points' x and y coordinates. This means the parent and path length lookup tables will be populated in $O(dist^2 P)$ time. Iterating the points in longest-path-first order requires them to be sorted, taking $O(P \log P)$ time. Lastly, each point has only one parent, so tracing back the chains takes $O(P)$ time. Overall, mining synteny blocks from the set of points takes $O(\max(dist^2 P, P \log P))$ time.

4.6.7 Track Packing

The extended Repeat and multiple alignment algorithms detect inversions in micro-synteny search results so that they may be explicitly drawn. In cases where more than one inversion has been found in a single alignment, all inversions are to be

drawn as compactly as possible, that is, we want to pack as many inversions into as few visualization rows as possible without overlapping any of the inversions. Similarly, in the search view's macro-synteny visualization we want to draw the synteny blocks from the same chromosome as compactly as possible.

This "Track Packing" problem is equivalent to the Interval Partitioning/Coloring (IP) problem for which there exists an optimal polynomial-time solution [52]. The solution works by iteratively applying the greedy polynomial-time solution for the Interval Scheduling (IS) problem [52] to a set of intervals until all the intervals have been scheduled. In the case of the micro-synteny alignments, the intervals are the position intervals described in Section 4.6.3. In the case of the macro-synteny blocks, the intervals are the blocks' genomic positions on the reference chromosome.

4.6.7.1 Time Complexity

Let I be the total number of intervals to be packed. The IS problem requires that the intervals first be sorted by end time, taking $O(I \log I)$ time. It then iterates the intervals and greedily adds them to the schedule in $O(I)$ time. The IP problem repeats this iteration step until each interval has been added to a schedule. In the worst case, all the intervals will overlap and so this step will take I iterations, yielding a time complexity of $O(I^2)$. In our application, this worst case scenario is highly unlikely, so we expect the sorting to dominate the run-time.

4.7 Experimental Results

In this section we investigate the efficacy of using gene family assignments as a surrogate for direct pairwise sequence comparison by computing whole chromosome synteny blocks at the genic resolution. We then report on how the GCV is currently being used by the Legume Information System and Legume Federation projects.

4.7.1 Block Resolution

DAGchainer [34] is an algorithm commonly used to compute "chains" of syntenic genes (synteny blocks) in complete genomes via dynamic programming. It takes as input a list of homologous gene pairs, typically computed by a pairwise sequence comparison method, such as the Basic Local Alignment Search Tool (BLAST) [2]. MCScanX [109] is an algorithm commonly used to identify collinear segments (syntenic blocks) in multiple genomes or highly redundant genomes, such as legumes. MCScanX works by computing pairwise segments of collinearity using DAGchainer and then combines them into larger segments by identifying overlapping related segments while resolving ambiguities caused by structural events, such as rearrangements, duplications, and inversions.

We compared the results of MCScanX on our gene family assignment derived homologous groups to the results of basic MCScanX on pairwise BLASTs of the coding DNA sequence (CDS) for the same genome. For a self-comparison of the *Glycine max* (soybean) genome, the latter puts 66% of all gene models into 1110 blocks ranging from 5 to 1074 genes in extent. Using an unfiltered set of gene families, the comparable procedure yields 73% of genes in 2825 blocks ranging in size from 5 to 1102 genes. Inspection of the blocks produced with this approach suggested that many of the excess blocks were the results of genes from massively expanded families residing in large clusters. Since our algorithms for finding similar regions require that the families matched be distinct, we repeated the comparison by filtering (masking) our families to exclude from consideration any that contained more than 20 members from soybean. The results of this procedure were more similar to the BLAST-based approach with 61% of genes placed into 1319 blocks ranging in size from 6 to 983 genes. Most of these blocks are coequal in extent to the corresponding blocks from the BLAST-based procedure, but are missing some internal anchor genes due to the

pre-filtering procedure; in GCV's micro-synteny search implementation, these high copy gene families would still be scored as matches during the alignment procedure used on the resulting blocks. Furthermore, the macro-syntenic blocks produced by MCScanX on the gene family assignments are quite similar to those precomputed by a pipeline based on DAGchainer and used in the Legume Information System implementation of the GCV (Figure 4.7).

4.7.2 LIS and LegFed Integration

The Legume Information System[9] (LIS) [20] is an online platform for legume breeders and researchers that houses a variety of genetic and genomic data of model legume species relevant to industrial agriculture. The Legume Federation[10] (LegFed) is a consortium of legume researchers and groups, including LIS, with the objective of fostering the adoption of data standards, distributed development, and enabling comparative analyses. GCV was born from the needs of these projects and is integrated into the sites as follows.

LIS acquires annotated genomes from a variety of independently managed projects and primary data repositories. In accordance with the mission of the Legume Federation, homology relationships among the genes annotated in these genomes are established by initial HMM-based assignment to the Phytozome Angiosperm-level gene families [32]; this in itself is sufficient for the purposes of making genomes available for use in GCV. Further steps of multiple sequence alignment and phylogenetic gene tree construction are used to produce interactive tree visualizations which interoperate with GCV in several ways. For example, a collection of genes can be specified from an arbitrary subtree and loaded into the multi view of GCV as the

[9] http://legumeinfo.org/lis_context_viewer
[10] http://legumefederation.org/lis_context_viewer

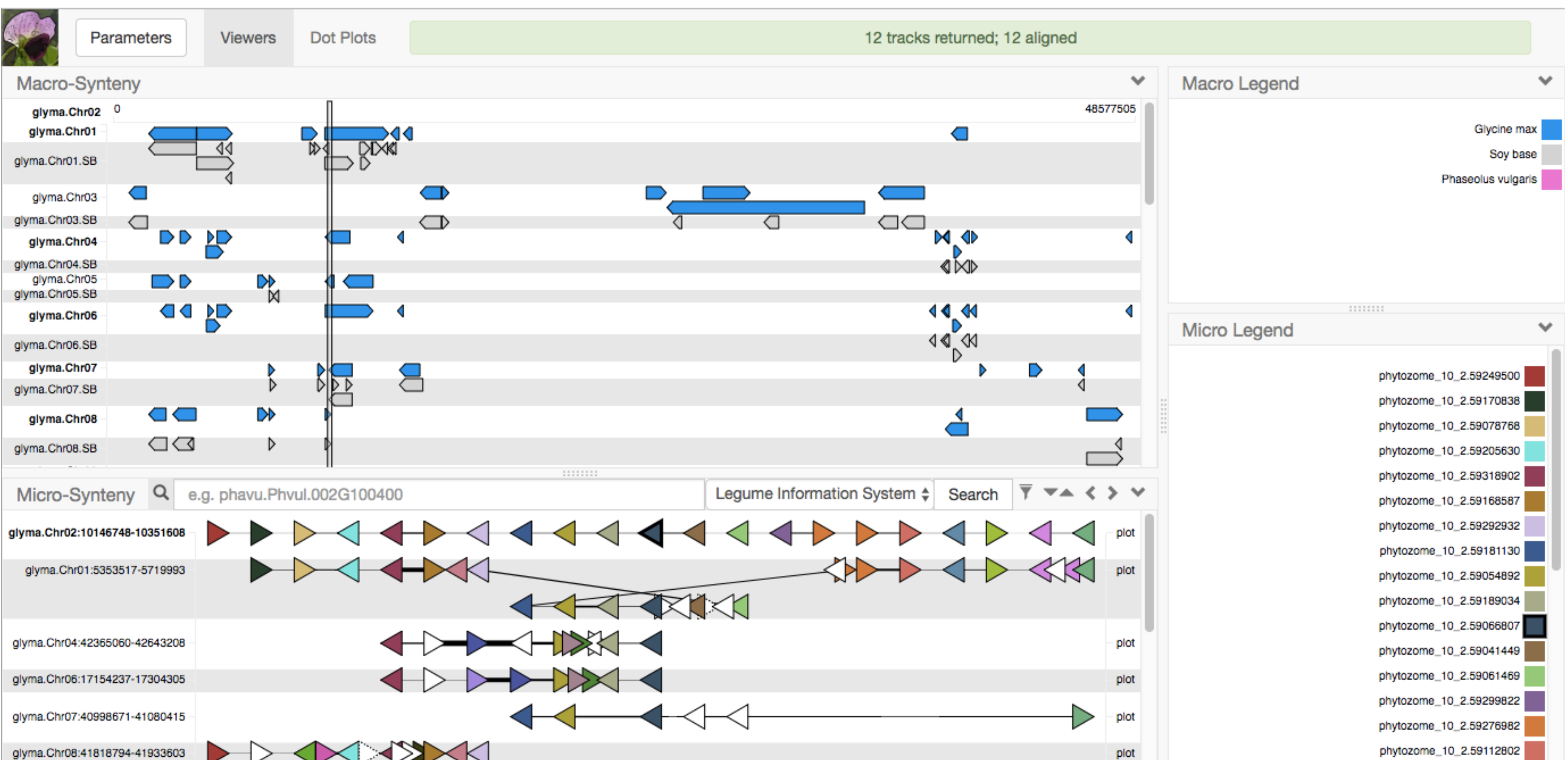

Figure 4.7: Comparison of results from MCScanX using gene family assignments (blue blocks) to those produced using DAGchainer on CDS pairwise matches (gray blocks). As can be seen, the results are generally similar, with the gene-family based use of MCScanX tending to coalesce into larger blocks some regions called as fragmentary by DAGchainer. It can also be seen that the GCV algorithm for determining micro-synteny to the region of soybean chromosome 2 used as the query is producing results of comparable sensitivity to those produced by MCScanX, as evidenced by the small blocks from chromosomes 4 and 6 found in both the MCScanX macro-synteny blocks and the GCV micro-synteny representation, but missing from the DAGchainer-based results. On the other hand, MCScanX tends to ignore inverted segments that disrupt larger collinear blocks, as displayed in the focus region of the example and present in the corresponding DAGchainer blocks. See Figure 4.4 for the effects of this behavior in the context of the PGDD implementation of block search and display.

set of focus genes from which context tracks are derived. Alternatively, a leaf node of a phylogeny can be used as the focus gene of a query track in the search view of GCV. This linking can also be reversed, that is, in GCV the user may select a gene family and view the phylogenetic tree for that family with the members present in the linking GCV context highlighted.

Given the vast amount and different types of data housed by LIS, a heterogeneous collection of software is used to facilitate user exploration and knowledge discovery. As illustrated by the interoperability of the phylogenies and GCV, interlinking between these software is crucial to the utility of LIS. As such, GCV is configured to link to various LIS tools including Tripal gene pages [29] and InterMine reports [101]. Additionally, a service is used to link to external tools specific to the sources from which the various genomes were acquired, as described in Section 4.5.2.

4.8 Conclusions

The GCV is a powerful tool that is already being used by researchers to investigate functional contexts of interest, assess assembly/annotation quality, and perform analyses on geographically distributed data. By utilizing a federated data model that only requires a common set of gene family annotations among data providers, the GCV is highly scalable and may be used to perform analyses on large taxonomic groups that are beyond the current capabilities of pan-genomic methods operating at the DNA resolution.

CHAPTER FIVE

CONCLUSION AND FUTURE WORK

In Chapter 1, we defined the term "pan-genomics" and posed a set of research questions concerning the challenges of pan-genomics from a computational perspective. Then, in Chapters 2 through 4, we addressed these questions with a variety of algorithms, data-structures, and software. Specifically, in Chapters 2 and 3, we contributed methods for mining synteny from pan-genomic de Bruijn graphs, with applications to real data sets and explorations in visualization and machine learning (see Contributions 1 and 2 in Chapter 1). And in Chapter 4, we contributed methods for mining synteny and structural variation at the genic level across large taxonomic groups, with an emphasis on data federation and visualization (see Contribution 3 in Chapter 1). Though these methods have been shown effective, there is still much room for improvement. Here, we briefly discuss such improvements and possible future directions.

5.1 Future Work

Computational pan-genomics is still a young field with many open problems. Here, we discuss how we can extend the work of this dissertation to further address these problems and discuss additional directions that complement this work and address other open problems.

5.1.1 Identifying Interesting Biology

In Chapters 2 and 3, we gave specific examples of interesting biology that was captured with our Approximate Frequent Subpaths (AFS) and Frequented Regions (FR) algorithms, specifically, the HFI1 gene in Chapter 2 and three EC1118

insertions in Chapter 3. In the case of the HFI1 genes, the gene was found by manually identifying AFSs that had different copy numbers in different accessions. After sorting all such AFSs from longest to shortest (in nucleotides), the HFI1 gene was found because it corresponded to the longest AFS. In the case of the EC1118 insertions, we were specifically interested in whether these insertions were captured in the FRs we mined, so we examined FRs that were supported by EC1118 and overlapped with the regions that the insertions were known to occur in. Though these manual workflows are effective, they are also tedious. For this reason, we would like to develop semi-automated processes to help identify potentially relevant biology in a set of AFSs/FRs.

As we discussed in Chapters 2 and 3, the leniency of the algorithm parameters can affect the coherence of the results, but it is ultimately the user's domain knowledge that determines whether a result is interesting or not, despite the algorithm parameters. So what constitutes interesting biology depends on the specific interests of the user. It is hard to conceive how a user would convey this to our algorithms; a more reasonable method of empowering the user would be to provide them a means of interacting with the results to discover the insights they seek. This could be done through a tool (command line or graphical user interface) that allows them to interactively search and filter the results on a variety of metrics, for instance, by what strains support an FR, the number of supporting paths an FR has, the percent identify of an FR's supporting paths, whether or not an FR lies in a coding region of interest. This could include automated post-analyses as well, such as the identification of specific structural events: insertions, copy number variation, etc. By allowing users to combine and compose such interactions, they should be able to rapidly explore the results and glean a far broader set of results than those we have reported here.

5.1.2 Furthering Frequented Regions

In Chapter 3, we introduced the FR problem and an efficient algorithm, `FindFRs`, based on agglomerative clustering. We also introduced a parallelized version of the algorithm based on computing a graph matching and showed that the algorithm was scalable, both in terms of the number of threads and the size of the graph, suggesting that the algorithm can scale to many more threads on much larger data sets. For this reason, we want to further improve the utility of the algorithm by providing implementations that target the GPU and cluster computing paradigms, for which variations of the matching algorithm we implemented already exist [3, 38]. By providing such implementations, users will be able to mine FRs from pan-genomic data much more quickly on a single computer by leveraging their GPU and from much larger populations that would otherwise be intractable for a single computer.

Though effective for analyzing compressed de Bruijn graphs (CDBGs), we argued in Chapter 4 that the agglomerative nature of the algorithm made it insufficient for annotation graphs, specifically, gene family graphs, because the gene family alphabet is much smaller than the k-mer alphabet, meaning a gene family is much more likely to participate in multiple FRs than a k-mer. For this reason, we introduced a "fuzzy" variation of the FR algorithm by simply repeatedly applying `FindFRs` and keeping only the largest FR from each application. Though effective for the specific use case of Chapter 4, this would not work at scale for two reasons: 1) FRs that will be included in the result set cannot be identified in parallel, and 2) for genome paths to participate in multiple FRs, only the subpath that supports an FR will be removed at each application, rather than the entire path, meaning the time complexity will be much worse than that reported in Chapter 4.

A more reasonable approach to making `FindFRs` fuzzy is splitting vertices when they are merged into a cluster. Specifically, if a vertex has sufficient (but less) support

to be merged with a cluster other than the one it is being merged with, it should be split into two separate vertices, one to participate in the impending merge and one to remain and be merged with another cluster. Though the details of how specifically a split would be performed remain unclear, the approach would not conflict with any of the algorithm properties necessary to achieve parallelism, and so would fit nicely into the existing algorithm. Furthermore, the membership degree of a vertex (discussed in Section 4.6.4 of Chapter 4) can be leveraged during analyses, such as by the method outlined in Section 5.1.1.

An alternative approach to hierarchical fuzzy clustering that we have explored as a means of identifying graph regions that are frequent but is not solving the FR problem per se is spectral graph analysis. It has been shown that the spectrum of a graph can be used to reveal the hierarchical community structure within a graph [108]. This information can then be used to guide the repeated application of spectral clustering to the graph to construct a fuzzy community hierarchy. We have devised a method of encoding chromosomes represented as ordered sequences of gene families into a path graph such that this method of spectral graph analysis can be used to mine fuzzy hierarchies of subpaths that are approximately the same within two or more of the chromosomes. Preliminary experiments on a subset of the chromosomes from the Legume Information System (LIS) from Chapter 4 indicate that this method is effective. Unfortunately, computing the spectrum of a graph is computationally expensive, and our encoding represents a gene family with a multiplicity of vertices, once for each time it occurs in a chromosome, meaning the method is impractical unless a more efficient encoding can be devised.

Despite the current shortcomings, graph spectral analysis is an alluring method for the graph path analysis of pan-genomic data because it can serve as a gateway to performing analyses on pan-genomic data that have been projected into a lower

dimensional space. This is especially relevant when considering the ever-growing volume of genomic data and the algorithmic paradigms that will need to be leveraged in order to analyze them, such as the federated approach described in Chapter 4.

In Chapter 3, we also introduced machine learning techniques for classifying genomes based on their FR content. Though we showed that Support Vector Machines (SVMs) were effective for the task, there is still room for improvement, especially for CDBGs with smaller k-values. Two potential research directions are: 1) devising SVM kernels specifically for FR-based classification problems, and 2) exploring other supervised learning techniques. Regarding the latter, we are specifically interested in using a random forest as our classification model. This is because, unlike SVMs, random forests are intrinsically suited for multi-class classification problems and work well with a mixture of numerical and categorical features [37], meaning we could consider additional data when performing our analyses, such as gene annotations. Furthermore, random forests are also resistant to over-fitting.

5.1.3 Extending the Genome Context Viewer

In Chapter 4, we introduced the Genome Context Viewer (GCV). Though the primary function of the GCV is the visual exploration of micro-syntenic contexts, it would be useful if the user could also compare homologous genes at a finer granularity, providing them with information about structural preservation and variation within the genes, such as the conservation of ancestral exon boundaries. For websites built on a Chado database, such as LIS, these data are already available and simply need endpoints in the GCV's RESTful API. The data could then be visualized by adding another genome browser style viewer to the UI, such as Aequatus [105], an open-source homology browser built with similar Web technologies as the GCV that is specifically designed for comparing structure between multiple genes.

Another shortcoming of the GCV is that it is not trivial for a user to deploy an instance locally. In this regard, we aim to make the GCV more like the Genome Analysis Toolkit [73] – a fully integrated pipeline that takes raw sequence reads as input, assembles them into genomes, and then performs various analyses. Though we are not interested in taking raw reads as input, having a well defined pipeline that consumes common bioinformatic data types and converts them into data that can be used by the GCV would greatly expand its accessibility to less technically savvy users.

Lastly, we want to further improve the GCV by extending it to operate on generic sequences, not just sequences of genes with gene family annotations. This could allow it to, for example, analyze and visualize chromosomes as sequences of FRs or de Bruijn graph nodes. To this end, we also want it to be able to handle graphical representations of the input data. For example, in [78] we presented an interactive viewer that represents a pan-genome as an FR graph, similar to those depicted in Figures 3.4 and 3.5 in Chapter 3. With these extensions, the GCV could be leveraged in a larger range of analyses, making it relevant to a broader set of users.

5.1.4 Pan-Genome Representation

An issue that is not apparent in this manuscript but is a need of the pan-genomics community is better methods for constructing pan-genomic CDBGs. In Chapter 3, we used [5] to construct the CDBGs we analyzed. This is currently the fastest and most memory efficient pan-genome CDBG construction method available. Even so, it takes several hours to construct the $k = 25$ CDBG for 55 strains of yeast and cannot run to completion even on a small subset of the LIS genomes. This raises two questions: 1) How do we improve upon the existing state of the art, and 2) is there an alternative representation of pan-genome data that would be fundamentally more

scalable than the de Bruijn based approach?

Regarding (1), parallelization of the existing construction algorithms would be the first step. As [5] is based on the Burrows-Wheeler transform (BWT), there is a wealth of existing parallel BWT based methods specifically for DNA that could serve as a starting point, more recently [63] and [65]. As for (2), we showed in Chapters 2 and 3 that the de Bruijn graph is a structure that lends itself to pan-genomic analysis. However, a shortcoming that we repeatedly emphasized is the selection of an appropriate k-value. Currently, there is no best practice for this task. In [13], we explored using persistent homology to determine a set of k-values that captured graph structures (bubbles) that were persistent relative to the spectrum of possible k-values, but the approach lacked stability when minor permutations were made to the data from which the graph was derived. To our knowledge, we are the first to try and address this problem, and the solution still remains elusive.

Given the k-value issue, it is worth considering alternative data structures that are not only more scalable, but can represent multiple k-values or, better yet, have no notion of a k-value, but rather, significant structure. Since the k-value is intimately connected to the structure of the de Bruijn graph, we would have to distill the properties of the graph that best characterize the population, as we attempted to do in [13]. Revealing these properties may be a matter of better understanding the class of data structures we are considering for pan-genome analysis. To this end, a good starting point may be Wheeler Graphs [30] – a framework that describes deep relationships among variations of the BWT (including de Bruijn graphs) and provides a foundation for devising new variations. Furthermore, it would be ideal if the data structure were capable of representing additional, non-genetic information, such as gene annotations and quantitative trait loci.

5.1.5 Additional Directions

In this work, we primarily focused on the problem of identifying synteny in a population of genomes, both at the DNA and genic levels. Though identifying synteny is a fundamental biological analysis, there are other analyses worth pursing in the context on pan-genomics. One such analysis is computing a genome that best represents all the genomes in a population. This could be used as a non-biased reference when performing traditional (reference-centric) bioinformatic analyses and assembling new genomes. Currently, there are two techniques for computing such a genome: In [22], the authors represent the population using a population graph, where all genomes in the population are aligned to a reference and bubbles are added where variations occur. New genomes, including the most representative, are then constructed from this graph using a hidden Markov model. In [46], the authors construct a representative genome by iteratively aligning genomes to the current representative and then updating it based on the new alignment. Both approaches are biased, the first by the use of a reference, and the second by the order in which the genomes are aligned to the representative. This illustrates the need for a method that is inherently unbiased while allowing for genomes to be added or removed from the population at a later time.

Another challenge that needs to be addressed is defining an unambiguous coordinate system for pan-genomes. When sequences are aligned to a reference, the location of their variations can be described in terms of coordinates on the reference genome. Since a pan-genome does not have a reference, such methods fail. One option is to define all feature coordinates relative to each genome that participates in the feature, as we have done with our software in Chapters 2 and 3. Though effective, this method is not succinct. In fact, the coordinate information rapidly grows unwieldy relative to the number of genomes participating in a feature. An

alternative worth pursing is to define a projection of a pan-genome into a normalized space, that is, the properties of the space are invariant relative to the structure of the pan-genome, so while the contents of the population may change, the coordinates will remain the same. The crux of this approach is to define a projection that yields intuitive coordinates.

A related problem is defining distances between pan-genomic features. Again, since there is no notion of a reference and features of interest may span several genomes, a projection may go a long way towards quantifying distances in pan-genomic space.

Lastly, there is a need for file formats that can succinctly represent pan-genomic data while integrating well with existing community standards. The CDBG construction methods currently available [5, 71, 96] produce DOT (graph description language) files, which can represent (un)directed graphs with arbitrary attributes. Though expressive, this format does not intuitively support sequence, genomic feature, annotation, or alignment information, as do the FASTA/FASTQ [14, 87], BED [91], GO [15], and SAM/BAM [4, 60] file formats, respectively. In order to be widely adopted by the bioinformatics community, pan-genomic data needs to be represented using a domain-specific file format that complements other existing bioinformatic file formats. Furthermore, given the potentially vast quantity of data that composes a pan-genome, the format should be compressible while supporting rapid search and efficient decompression of subsets of the data, similar to the BAM format.

5.2 Closing Remarks

As the cost of sequencing DNA continues to drop, the number of sequenced genomes continues to rapidly grow, both within and among related species. This

necessitates the need for data structures, algorithms, and statistical methods to perform bioinformatic analyses of pan-genomic data. In Chapter 1, we characterized this need with a set of research questions. In Chapters 2 and 3, we addressed these questions with novel methods for mining synteny from pan-genomic de Bruijn graphs by considering the Approximate Frequent Subpaths and Frequented Regions problems as a means of mining syntenic blocks. We then explored a variety of analyses that mining synteny blocks from pan-genomic de Bruijn graphs enables, including pan-genome visualization and machine-learning. In Chapter 4, we presented a novel interactive data mining tool — the Genome Context Viewer — for mining syntenic blocks at the genic resolution from data distributed across a heterogeneous set of data providers. By using gene family annotations as a unit of search and comparison, the Genome Context Viewer is able to perform traditionally cumbersome analyses on-demand in a federated manner.

These methods are already being used by crop breeders and researchers to better understand and improve crops relevant to industrial agriculture. By pursing the work outlined in this Chapter, these methods will be further improved, enhancing their utility and making them more broadly applicable to other industries and fields of research.

REFERENCES CITED

[1] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules. In *Proceedings of VLDB* (1994), vol. 1215, pp. 487–499.

[2] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic local alignment search tool. *Journal of Molecular Biology 215*, 3 (1990), 403–410.

[3] Auer, B. O. F., and Bisseling, R. H. A GPU algorithm for greedy graph matching. In *Facing the Multicore-Challenge II*. Springer, 2012, pp. 108–119.

[4] Barnett, D. W., Garrison, E. K., Quinlan, A. R., Strömberg, M. P., and Marth, G. T. BamTools: a C++ API and toolkit for analyzing and managing BAM files. *Bioinformatics 27*, 12 (2011), 1691–1692.

[5] Beller, T., and Ohlebusch, E. Efficient construction of a compressed de Bruijn graph for pan-genome analysis. In *Combinatorial Pattern Matching* (2015), Springer, pp. 40–51.

[6] Bentley, J. L. Survey of techniques for fixed radius near neighbor searching. Tech. Rep. SLAC-186, Stanford Linear Accelerator Center, Calif.(USA), 1975.

[7] Bewick, V., Cheek, L., and Ball, J. Statistics review 13: receiver operating characteristic curves. *Critical Care 8*, 6 (2004), 508.

[8] Borneman, A. R., Desany, B. A., Riches, D., Affourtit, J. P., Forgan, A. H., Pretorius, I. S., Egholm, M., and Chambers, P. J. Whole-genome comparison reveals novel genetic elements that characterize the genome of industrial strains of Saccharomyces cerevisiae. *PLoS Genetics 7*, 2 (2011), e1001287.

[9] Bradnam, K. R., Fass, J. N., Alexandrov, A., Baranay, P., Bechner, M., Birol, I., Boisvert, S., Chapman, J. A., Chapuis, G., Chikhi, R., Chitsaz, H., Chou, W.-C., Corbeil, J., Del Fabbro, C., et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience 2*, 1 (2013), 1–31.

[10] Bridges, M., Heron, E. A., O'Dushlaine, C., Segurado, R., Morris, D., Corvin, A., Gill, M., Pinto, C., and Consortium, I. S. Genetic classification of populations using supervised learning. *PloS One 6*, 5 (2011), e14802.

[11] Cheng, H., Yu, P. S., and Han, J. Ac-close: Efficiently mining approximate closed itemsets by core pattern recovery. In *Proceedings of ICDM'06* (2006), pp. 839–844.

[12] Clamp, M., Cuff, J., Searle, S. M., and Barton, G. J. The Jalview Java alignment editor. *Bioinformatics 20*, 3 (2004), 426–427.

[13] Cleary, A., Fasy, B., Ramaraj, T., Mudge, J., and Mumey, B. Pangenomics: Persistent homology for pan-genome analysis. In *11th Annual Sequencing, Finishing, and Analysis in the Future Meeting (LANL SFAF)* (2016).

[14] Cock, P. J., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research 38*, 6 (2009), 1767–1771.

[15] Consortium, G. O. The Gene Ontology (GO) database and informatics resource. *Nucleic Acids Research 32*, suppl_1 (2004), D258–D261.

[16] Consortium, I. H. G. S. Initial sequencing and analysis of the human genome. *Nature 409*, 6822 (2001), 860.

[17] Consortium, T. C. P.-G. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics 19*, 1 (2018), 118–135.

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

[19] Cortes, C., and Vapnik, V. Support-vector networks. *Machine Learning 20*, 3 (1995), 273–297.

[20] Dash, S., Campbell, J. D., Cannon, E. K., Cleary, A. M., Huang, W., Kalberer, S. R., Karingula, V., Rice, A. G., Singh, J., Umale, P. E., Weeks, N. T., Wilkey, A. P., Farmer, A. D., and Cannon, S. B. Legume information system (legumeinfo.org): a key component of a set of federated data resources for the legume family. *Nucleic Acids Research 44*, D1 (2016), D1181–D1188.

[21] de Bruijn, N., and Erdös, P. On a combinatorial problem. In *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam* (1948), vol. 49, pp. 1277–1279.

[22] Dilthey, A., Cox, C., Iqbal, Z., Nelson, M. R., and McVean, G. Improved genome inference in the MHC using a population reference graph. *Nature Genetics 47*, 6 (2015), 682.

[23] Dunham, M. J., Badrane, H., Ferea, T., Adams, J., Brown, P. O., Rosenzweig, F., and Botstein, D. Characteristic genome rearrangements in experimental evolution of Saccharomyces cerevisiae. *Proceedings of National Academy of Sciences 99*, 25 (2002), 16144–16149.

[24] Dunn, B., Richter, C., Kvitek, D. J., Pugh, T., and Sherlock, G. Analysis of the Saccharomyces cerevisiae pan-genome reveals a pool of copy number variants distributed in diverse yeast strains from differing industrial environments. *Genome Research 22*, 5 (2012), 908–924.

[25] Dunn, J. C. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics 3*, 3 (1973), 32–57.

[26] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[27] Edgar, R. C. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research 32*, 5 (2004), 1792–1797.

[28] Edmonds, J. Paths, trees, and flowers. *Canadian Journal of Mathematics 17*, 3 (1965), 449–467.

[29] Ficklin, S. P., Sanderson, L.-A., Cheng, C.-H., Staton, M. E., Lee, T., Cho, I.-H., Jung, S., Bett, K. E., and Main, D. Tripal: a construction toolkit for online genome databases. *Database 2011* (2011), bar044.

[30] Gagie, T., Manzini, G., and Sirén, J. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science 698* (2017), 67–78.

[31] Good, I. J. Normal recurring decimals. *Journal of the London Mathematical Society 1*, 3 (1946), 167–169.

[32] Goodstein, D. M., Shu, S., Howson, R., Neupane, R., Hayes, R. D., Fazo, J., Mitros, T., Dirks, W., Hellsten, U., Putnam, N., and Rokhsar, D. S. Phytozome: a comparative platform for green plant genomics. *Nucleic Acids Research 40*, D1 (2012), D1178–D1186.

[33] Guha, S. Efficiently mining frequent subpaths. In *Proceedings of the Australasian Data Mining Conference* (2009), pp. 11–15.

[34] Haas, B. J., Delcher, A. L., Wortman, J. R., and Salzberg, S. L. DAGchainer: a tool for mining segmental genome duplications and synteny. *Bioinformatics 20*, 18 (2004), 3643–3646.

[35] Higgins, D. G., and Sharp, P. M. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene 73*, 1 (1988), 237–244.

[36] Hill, S., Srichandan, B., and Sunderraman, R. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *Proceedings of ACM BCB* (2012), pp. 661–666.

[37] Ho, T. K. Random decision forests. In *Proceedings of the IEEE Conference on Document Analysis and Recognition* (1995), vol. 1, pp. 278–282.

[38] Hoepman, J. Simple distributed weighted matchings. *CoRR cs.DC/0410047* (2004).

[39] Hogeweg, P., and Hesper, B. The alignment of sets of sequences and the construction of phyletic trees: an integrated method. *Journal of Molecular Evolution 20*, 2 (1984), 175–186.

[40] Huan, J., Wang, W., Bandyopadhyay, D., Snoeyink, J., Prins, J., and Tropsha, A. Mining protein family specific residue packing patterns from protein structure graphs. In *Proceedings of RECOMB* (2004), pp. 308–315.

[41] Idury, R. M., and Waterman, M. S. A new algorithm for DNA sequence assembly. *Journal of Computational Biology 2*, 2 (1995), 291–306.

[42] Inokuchi, A., Washio, T., and Motoda, H. An apriori-based algorithm for mining frequent substructures from graph data. In *European Conference on Principles of Data Mining and Knowledge Discovery* (2000), Springer, pp. 13–23.

[43] Institute, N. H. G. R. DNA sequencing costs. `http://www.genome.gov/sequencingcosts/`, 2017. Accessed: 2018-3-10.

[44] Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics 44*, 2 (2012), 226–232.

[45] Ishtar Snoek, I., and Yde Steensma, H. Factors involved in anaerobic growth of saccharomyces cerevisiae. *Yeast 24*, 1 (2007), 1–10.

[46] Jandrasits, C., Dabrowski, P. W., Fuchs, S., and Renard, B. Y. seq-seq-pan: Building a computational pan-genome data structure on whole genome alignment. *BMC Genomics 19*, 1 (2018), 47.

[47] Jiang, C., Coenen, F., and Zito, M. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review 28*, 01 (2013), 75–105.

[48] Joachims, T. Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning* (1998), Springer, pp. 137–142.

[49] Kao, K. C., and Sherlock, G. Molecular characterization of clonal interference during adaptive evolution in asexual populations of Saccharomyces cerevisiae. *Nature Genetics 40*, 12 (2008), 1499–1504.

[50] Kim, J., Larkin, D. M., Cai, Q., Asan, E., Zhang, Y., Ge, R.-L., Auvil, L., Capitanu, B., Zhang, G., Lewin, H. A., and Ma, J. Reference-assisted chromosome assembly. *Proceedings of the National Academy of Sciences 110*, 5 (2013), 1785–1790.

[51] Kingsford, C., Schatz, M. C., and Pop, M. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics 11*, 1 (2010), 21.

[52] Kleinberg, J., and Tardos, E. *Algorithm design.* Addison-Wesley, 2005.

[53] Koyutürk, M., Grama, A., and Szpankowski, W. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics 20*, suppl 1 (2004), i200–i207.

[54] Krogh, A., Brown, M., Mian, I. S., Sjölander, K., and Haussler, D. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology 235*, 5 (1994), 1501–1531.

[55] Kruskal, J. B., and Wish, M. *Multidimensional scaling*, vol. 11. Sage, 1978.

[56] Krzywinski, M., Schein, J., Birol, I., Connors, J., Gascoyne, R., Horsman, D., Jones, S. J., and Marra, M. A. Circos: an information aesthetic for comparative genomics. *Genome Research 19*, 9 (2009), 1639–1645.

[57] Lance, G. N., and Williams, W. T. Computer programs for hierarchical polythetic classification ("similarity analyses"). *The Computer Journal 9*, 1 (1966), 60–64.

[58] Lance, G. N., and Williams, W. T. Mixed-data classificatory programs I - agglomerative systems. *Australian Computer Journal 1*, 1 (1967), 15–20.

[59] Lee, T.-H., Kim, J., Robertson, J. S., and Paterson, A. H. Plant genome duplication database. *Plant Genomics Databases: Methods and Protocols* (2017), 267–277.

[60] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., and Durbin, R. The sequence alignment/map format and SAMtools. *Bioinformatics 25*, 16 (2009), 2078–2079.

[61] Li, H., and Homer, N. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics 11*, 5 (2010), 473–483.

[62] Li, R., and Wang, W. REAFUM: Representative approximate frequent subgraph mining. In *SIAM International Conference on Data Mining* (2015), pp. 2167–0099.

[63] Liu, B., Zhu, D., and Wang, Y. deBWT: parallel construction of Burrows–Wheeler transform for large collection of genomes with de Bruijn-branch encoding. *Bioinformatics 32*, 12 (2016), i174–i182.

[64] Liu, J., Paulsen, S., Sun, X., Wang, W., Nobel, A., and Prins, J. Mining approximate frequent itemsets in the presence of noise: Algorithm and analysis. In *SDM* (2006), vol. 6, pp. 405–416.

[65] Liu, Y., Hankeln, T., and Schmidt, B. Parallel and space-efficient construction of Burrows-Wheeler transform and suffix array for big genome data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) 13*, 3 (2016), 592–598.

[66] Lopez, M. D., and Samuelsson, T. eGOB: eukaryotic gene order browser. *Bioinformatics 27*, 8 (2011), 1150–1151.

[67] Lopez-Maestre, H., Brinza, L., Marchet, C., Kielbassa, J., Bastien, S., Boutigny, M., Monnin, D., Filali, A. E., Carareto, C. M., Vieira, C., Picard, F., Kremer, N., Vavre, F., Sagot, M.-F., and Lacroix, V. SNP calling from RNA-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence. *Nucleic Acids Research 44*, 19 (2016), e148.

[68] Louis, A., Nguyen, N. T. T., Muffato, M., and Crollius, H. R. Genomicus update 2015: KaryoView and MatrixView provide a genome-wide perspective to multispecies comparative genomics. *Nucleic Acids Research 43*, D1 (2015), D682–D689.

[69] Lyons, E., Pedersen, B., Kane, J., Alam, M., Ming, R., Tang, H., Wang, X., Bowers, J., Paterson, A., Lisch, D., and Freeling, M. Finding and comparing syntenic regions among Arabidopsis and the outgroups papaya, poplar, and grape: CoGe with rosids. *Plant Physiology 148*, 4 (2008), 1772–1781.

[70] Manne, F., and Bisseling, R. H. A parallel approximation algorithm for the weighted maximum matching problem. In *International Conference on Parallel Processing and Applied Mathematics* (2007), Springer, pp. 708–717.

[71] Marcus, S., Lee, H., and Schatz, M. C. SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics 30*, 24 (2014), 3476–3483.

[72] McCallum, A., and Nigam, K. A comparison of event models for naive bayes text classification. In *AAAI-98 Workshop on Learning for Text Categorization* (1998), vol. 752, Citeseer, pp. 41–48.

[73] McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., and DePristo, M. A. The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research 20*, 9 (2010), 1297–1303.

[74] Minkin, I., Patel, A., Kolmogorov, M., Vyahhi, N., and Pham, S. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In *Algorithms in Bioinformatics*. Springer, 2013, pp. 215–229.

[75] Minkin, I., Pham, S., and Medvedev, P. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics 33*, 24 (2017), 4024–4032.

[76] Moore, G. Cramming more components onto integrated circuits. *Readings in Computer Architecture* (1965), 56.

[77] Mount, D. W. Bioinformatics: sequence and genome analysis. *Bioinformatics: Sequence and Genome Analysis* (2004).

[78] Mumey, B., Cleary, A., Ramaraj, T., Kahanda, I., Mudge, J., and Kulkarni, S. Exploring frequented regions in pan-genomic graphs. In *15th Annual Rocky Mountain Bioinformatics Conference (ROCKY)* (2017).

[79] Mungall, C. J., Emmert, D. B., and Consortium, T. F. A Chado case study: an ontology-based modular schema for representing genome-associated biological information. *Bioinformatics 23*, 13 (2007), i337–i346.

[80] Murphy, K. P., Weiss, Y., and Jordan, M. I. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence* (1999), UAI'99, pp. 467–475.

[81] Myers, E. W., and Miller, W. Optimal alignments in linear space. *Bioinformatics 4*, 1 (1988), 11–17.

[82] Noble, W. S. What is a support vector machine? *Nature Biotechnology 24*, 12 (2006), 1565–1567.

[83] Notredame, C., Higgins, D. G., and Heringa, J. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology 302*, 1 (2000), 205–217.

[84] Novo, M., Bigey, F., Beyne, E., Galeote, V., Gavory, F., Mallet, S., Cambon, B., Legras, J.-L., Wincker, P., Casaregola, S.,

and Dequin, S. Eukaryote-to-eukaryote gene transfer events revealed by the genome sequence of the wine yeast Saccharomyces cerevisiae EC1118. *Proceedings of the National Academy of Sciences 106*, 38 (2009), 16333–16338.

[85] of Scientists, G. K. C. Genome 10k: A proposal to obtain whole-genome sequence for 10000 vertebrate species. *Journal of Heredity 100*, 6 (2009), 659–674.

[86] Pagani, I., Liolios, K., Jansson, J., Chen, I.-M. A., Smirnova, T., Nosrat, B., Markowitz, V. M., and Kyrpides, N. C. The Genomes OnLine Database (GOLD) v. 4: status of genomic and metagenomic projects and their associated metadata. *Nucleic Acids Research 40*, D1 (2012), D571–D579.

[87] Pearson, W. R. *Rapid and sensitive sequence comparison with FASTP and FASTA*, vol. 183 of *Methods in Enzymology.* Academic Press, San Diego, 1990.

[88] Pevzner, P. A., Tang, H., and Waterman, M. S. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences 98*, 17 (2001), 9748–9753.

[89] Pham, S. K., and Pevzner, P. A. DRIMM-Synteny: decomposing genomes into evolutionary conserved segments. *Bioinformatics 26*, 20 (2010), 2509–2516.

[90] Preis, R. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS* (1999), vol. 99, Springer, pp. 259–269.

[91] Quinlan, A. R., and Hall, I. M. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics 26*, 6 (2010), 841–842.

[92] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2016.

[93] Rabey, H. A. E., Alshubaily, F., and Al-Otaibi, K. M. Phylogenetic relationships of some economically important cereal plants based on genome characterization using molecular markers. *Caryologia 68*, 3 (2015), 225–232.

[94] Ringnér, M. What is principal component analysis? *Nature Biotechnology 26*, 3 (2008), 303.

[95] Schütze, H., Manning, C. D., and Raghavan, P. *Introduction to Information Retrieval*, vol. 39. Cambridge University Press, 2008.

[96] Sheikhizadeh, S., Schranz, M. E., Akdel, M., de Ridder, D., and Smit, S. PanTools: representation, storage and exploration of pan-genomic data. *Bioinformatics 32*, 17 (2016), i487–i493.

[97] Shima, J., Ando, A., and Takagi, H. Possible roles of vacuolar H+-ATPase and mitochondrial function in tolerance to air-drying stress revealed by genome-wide screening of Saccharomyces cerevisiae deletion strains. *Yeast 25*, 3 (2008), 179–190.

[98] Siek, J. G., Lee, L.-Q., and Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[99] Sigaux, F. Cancer genome or the development of molecular portraits of tumors. *Bulletin de l'Academie Nationale de Medecine 184*, 7 (1999), 1441–1447.

[100] Skinner, M. E., Uzilov, A. V., Stein, L. D., Mungall, C. J., and Holmes, I. H. JBrowse: a next-generation genome browser. *Genome Research 19*, 9 (2009), 1630–1638.

[101] Smith, R. N., Aleksic, J., Butano, D., Carr, A., Contrino, S., Hu, F., Lyne, M., Lyne, R., Kalderimis, A., Rutherford, K., Stepan, R., Sullivan, J., Wakeling, M., Watkins, X., and Micklem, G. InterMine: a flexible data warehouse system for the integration and analysis of heterogeneous biological data. *Bioinformatics 28*, 23 (2012), 3163–3165.

[102] Smith, T. F., and Waterman, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1 (1981), 195–197.

[103] Stein, L. D., Mungall, C., Shu, S., Caudy, M., Mangone, M., Day, A., Nickerson, E., Stajich, J. E., Harris, T. W., Arva, A., and Lewis, S. The generic genome browser: a building block for a model organism system database. *Genome Research 12*, 10 (2002), 1599–1610.

[104] Tettelin, H., Masignani, V., Cieslewicz, M. J., Donati, C., Medini, D., Ward, N. L., Angiuoli, S. V., Crabtree, J., Jones, A. L., Durkin, A. S., DeBoy, R. T., Davidsen, T. M., Mora, M., Scarselli, M., et al. Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: Implications for the microbial "pan-genome". *Proceedings of the National Academy of Sciences 102*, 39 (2005), 13950–13955.

[105] Thanki, A. S., Ayling, S., Herrero, J., and Davey, R. P. Aequatus: An open-source homology browser. *bioRxiv* (2016), 055632.

[106] Vernikos, G., Medini, D., Riley, D. R., and Tettelin, H. Ten years of pan-genome analyses. *Current Opinion in Microbiology 23* (2015), 148–154.

[107] Viterbi, A. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory 13*, 2 (1967), 260–269.

[108] Wahl, S., and Sheppard, J. Hierarchical fuzzy spectral clustering in social networks using spectral characterization. In *FLAIRS Conference* (2015), pp. 305–310.

[109] Wang, Y., Tang, H., DeBarry, J. D., Tan, X., Li, J., Wang, X., Lee, T.-h., Jin, H., Marler, B., Guo, H., Kissinger, J. C., and Paterson, A. H. MCScanX: a toolkit for detection and evolutionary analysis of gene synteny and collinearity. *Nucleic Acids Research 40*, 7 (2012), e49.

[110] Wenger, J. W., Piotrowski, J., Nagarajan, S., Chiotti, K., Sherlock, G., and Rosenzweig, F. Hunger artists: yeast adapted to carbon limitation show trade-offs under carbon sufficiency. *PLoS Genetics 7*, 8 (2011).

[111] Yang, C., Fayyad, U., and Bradley, P. S. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *Proceedings of ACM SIGKDD* (2001), pp. 194–203.

[112] Yang, G. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of SIGKDD* (2004), pp. 344–353.

APPENDIX: SCHOLARLY PRODUCTS

Alan Cleary, Thiruvarangan Ramaraj, Indika Kahanda, Joann Mudge, and Brendan Mumey. Exploring frequented regions in pan-genomic graphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018. [journal]

Andrew Farmer, Jacqueline Campbell, Sudhansu Dash, Wei Huang, Malachy OConnell, Andrew Wilkey, Akshay Yadav, Nathan Weeks, Joel Berendzen, Alan Cleary, Sam Hokin, Connor Cameron, Vivek Krishnakumar, Amanda Cooksey, Agnes Chan, Ethalinda Cannon, Eric Lyons, Christopher Town, Steven Cannon, and David Fernandez-Baca. The federated plant database initiative for thee legumes. In *Plant and Animal Genome XXVI Conference (PAG)*, 2018. [poster]

Steven Cannon, Jacqueline Campbell, Sam Hokin, Joel Berendzen, Ethalinda Cannon, Alan Cleary, Sudhansu Dash, Connor Cameron, Wei Huang, Scott Kalberer, Amanda Cooksey, Nathan Weeks, and Andrew Farmer. legumeinfo.org: Legume research and trait data that is findable, accessible, interoperable and reusable. In *Plant and Animal Genome XXVI Conference (PAG)*, 2018. [poster]

Brendan Mumey, Alan Cleary, Thiruvarangan Ramaraj, Indika Kahanda, Joann Mudge, and Shubhang Kulkarni. Exploring frequented regions in pan-genomic graphs. In *15th Annual Rocky Mountain Bioinformatics Conference (ROCKY)*, 2017. [conference]

Alan Cleary and Andrew Farmer. Genome Context Viewer: visual exploration of multiple annotated genomes using microsynteny. *Bioinformatics*, 1:3, 2017. [journal]

Alan Cleary, Indika Kahanda, Brendan Mumey, Joann Mudge, and Thiruvarangan Ramaraj. Exploring frequented regions in pan-genomic graphs. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17, pages 89–97, 2017. [conference]

Alan Cleary, Brendan Mumey, Thiruvarangan Ramaraj, and Joann Mudge. Mining frequented regions for pan-genome analysis. In *12th Annual Sequencing, Finishing, and Analysis in the Future Meeting (LANL SFAF)*, 2017. [presentation]

Alan Cleary, Brendan Mumey, Thiruvarangan Ramaraj, and Joann Mudge. Approximate frequent subpath mining applied to pangenomics. In *International Conference on Bioinformatics and Computational Biology (BICoB)*, 2017. [conference]

Alan Cleary, Brendan Mumey, Thiruvarangan Ramaraj, and Joann Mudge. Investigating frequented regions (FRs) in a yeast pan-genome. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [poster]

Wei Huang, Sudhansu Dash, Alan Cleary, Alex Rice, Sam Hokin, Jacqueline Campbell, Joel Berendzen, Maria Hadres, Pooja Umale, Nathan Weeks, Andrew Wilkey, Andrew Farmer, Steven Cannon, and Ethalinda Cannon. PeanutBase.org: A genomic and genetic data resource to support crop improvement in peanut. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [poster]

Joel Berendzen, Ethalinda Cannon, Alan Cleary, Sudhansu Dash, Sam Hokin, Wei Huang, Alex Rice, Pooja Umale, Nathan Weeks, Andrew Wilkey, Andrew Farmer, Steven Cannon, and Maria Hadres. LegumeInfo.org: Legume research and trait data that is findable, accessible, interoperable and re-usable. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [poster]

Andrew Farmer, Alan Cleary, Sudhansu Dash, Alex Rice, Joel Berendzen, Sam Hokin, Maria Hadres, Pooja Umale, Jacqueline Campbell, Wei Huang, Nathan Weeks, Akshay Yadav, Andrew Wilkey, David Grant, Rex Nelson, Kevin Feeley, Victoria Carollo Blake, Ethalinda Cannon, Vivek Krishnakumar, Steven Cannon, Agnes Chan, Eric Lyons, Christopher Town, and David Fernandez-Baca. The Legume Information System (http://legumeinfo.org) and the Legume Federation (http://legumefederation.org): Comparative genomics and genetics through cooperative resource development. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [presentation]

Andrew Farmer and Alan Cleary. Genome Context Viewer: A user-friendly web application for visualizing and mining shared gene content among multiple genomes. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [poster]

Sudhansu Dash, Joel Berendzen, Jacqueline Campbell, Ethalinda Cannon, Steven Cannon, Alan Cleary, Andrew Farmer, Maria Hadres, Sam Hokin, Wei Huang, Alex Rice, Pooja Umale, Nathan Weeks, and Andrew Wilkey. Customizing Tripal sites with non-Tripal components at the Legume Information System and Peanutbase. In *Plant and Animal Genome XXV Conference (PAG)*, 2017. [presentation]

Alan Cleary, Thiruvarangan Ramaraj, Joann Mudge, and Brendan Mumey. Mining frequent subpaths in pan-genomes. In *11th Annual Sequencing, Finishing, and Analysis in the Future Meeting (LANL SFAF)*, 2016. [poster]

Alan Cleary, Brittany Fasy, Thiruvarangan Ramaraj, Joann Mudge, and Brendan Mumey. Pan-genomics: Persistent homology for pan-genome analysis. In *11th Annual Sequencing, Finishing, and Analysis in the Future Meeting (LANL SFAF)*, 2016. [poster]

Jacqueline Campbell, Sudhansu Dash, Ethalinda Cannon, Alan Cleary, Wei Huang, Scott Kalberer, Alex Rice, Jugpreet Singh, Pooja Umale, Nathan Weeks, Andrew Wilkey, Jeremy Town, Christopher DeBerry, David Fernandez-Baca, Andrew Farmer, and Steven Cannon. What's new in the Legume Information System and

the Federated Legume Database Initiative. In *Plant and Animal Genome XXIV Conference (PAG)*, 2016. [presentation]

Andrew Farmer, Alan Cleary, Alex Rice, Pooja Umale, Sam Hokin, Sudhansu Dash, Jacqueline Campbell, Wei Huang, Nathan Weeks, Andrew Wilkey, David Grant, Rex Nelson, Kevin Feeley, Vivek Krishnakumar, Akshay Yadav, Jeremy DeBerry, David Fernandez-Baca, Ethalinda Cannon, Christopher Town, and Steven Cannon. The Legume Information System and the Legume Federation: Working together for the legume-fed world. In *Plant and Animal Genome XXIV Conference (PAG)*, 2016. [presentation]

Sudhansu Dash, Wei Huang, Jacqueline Campbell, Jugpreet Singh, Alex Rice, Pooja Umale, Alan Cleary, Scott Kalberer, Nathan Weeks, Vijay Karingula, Prateek Gupta, Shivan Gunda, Ethalinda Cannon, Andrew Farmer, and Steven Cannon. LIS (Legume Information System): A clade based web resource for legumes. In *Plant and Animal Genome XXIV Conference (PAG)*, 2016. [poster]

Alan Cleary, Thiruvarangan Ramaraj, Joann Mudge, and Brendan Mumey. Mining frequent subpaths in pan-genomes. In *Montana State University Research Rendezvous*, 2016. [poster]

Sudhansu Dash, Jacqueline D. Campbell, Ethalinda K.S. Cannon, Alan M. Cleary, Wei Huang, Scott R. Kalberer, Vijay Karingula, Alex G. Rice, Jugpreet Singh, Pooja E. Umale, Nathan T. Weeks, Andrew P. Wilkey, Andrew D. Farmer, and Steven B. Cannon. Legume information system (legumeinfo.org): a key component of a set of federated data resources for the legume family. *Nucleic Acids Research*, 44(D1):D1181–D1188, 2016.

Alan Cleary, Longhui Ren, Steven Cannon, and Andrew Farmer. Tools for phylogenomic exploration within and among clade-oriented databases. In *Plant and Animal Genome XXIII Conference (PAG)*, 2015. [poster]

Steven Cannon, Andrew Farmer, Pooja Umale, Hrishikesh Lokhande, Alan Cleary, Nathan Weeks, Scott Kalberer, Ethalinda Cannon, Sudhansu Dash, Deepak Bitragunta, and Jugpreet Singh. The Legume Information System (LegumeInfo.org) 2015. In *Plant and Animal Genome XXIII Conference (PAG)*, 2015. [poster]

Andrew Farmer, Steven Cannon, Scott Kalberer, Jugpreet Singh, Ethalinda Cannon, Pooja Umale, Hrishikesh Lokhande, Alan Cleary, Nathan Weeks, Vijay Karingula, and Sudhansu Dash. Methods for collecting, integrating, and displaying complex genetic data, using the Legume Information System and PeanutBase. In *Plant and Animal Genome XXIII Conference (PAG)*, 2015. [presentation]