UNIVERSITY
OF OSLO

Master's thesis

KIVS - Graph **K**-mer **I**ndexer & **V**ariant **S**ignature Finder

Improving the performance of index creation for alignment-free genotyping

Sindre Ask Vestaberg

Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Sindre Ask Vestaberg

KIVS - Graph **K**-mer **I**ndexer & **V**ariant **S**ignature Finder

Improving the performance of index creation
for alignment-free genotyping

Supervisors:
Ivar Grytten
Knut Rand
Geir Kjetil

Abstract

Genotyping is the process of determining what genotypes (DNA sequences) an individual has at specific locations in the genome. The traditional approach to determine these genotypes is through variant calling. However, variant calling is computationally intensive as it requires the individual's genome to be aligned to a reference genome, which is an expensive process. Thus, alignment-free alternatives were developed that, while less accurate, are significantly faster than alignment-based methods by skipping the variant calling step. These alignment-free methods rely on identifying important k-mers (strings of k bases) for a species, to then look for these in individual genomes. These important k-mers are refered to as *variant signatures*, as they signify the presence of a variant. Finding these variant signatures requires computationally intensive preprocessing of data on known genetic variation for the species. For the human genome, the 1000 Genomes Project [1] provides this vast knowledge base on genetic variation to great benefit for alignment-free genotyping.

KAGE [13] is a recent and competitive alignment-free genotyper, both in terms of accuracy and speed. Compared to other existing solutions, such as Malva and PanGenie, KAGE is able to genotype both faster and more accurately. However, while KAGE has impressive performance when genotyping, this is not the case for the preprocessing of k-mers and variant signatures. Analyzing the vast amount of variant data to find and index all relevant k-mers is a time consuming process and makes it impractical to construct new indexes or update existing ones. As such, efficient solutions to these preprocessing steps would significantly improve the practicality of alignment-free solutions such as KAGE.

This thesis explores performance improvements for these preprocessing steps, resulting in KIVS, a high performance Python module for k-mer and variant signature analysis. KIVS achieves high performance and usability by being implemented in C++, wrapped in an easy-to-use Python interface. The genome and its possible variations are also represented by an optimized graph using 2-bit encoding to further improve performance. While made with KAGE integration in mind, KIVS is a standalone module that can be used by other genotyping implementations as well.

Acknowledgements

I would first and foremost like to thank my supervisors for their continued support and guidance while working on this thesis. Completing this project was made significantly easier thanks to them setting aside time for regular meetings, as well as providing quick answers and feedback to questions and drafts. Working alongside a fellow student whom shared my supervisors and field of study was also a great help.

Further I am also thankful for the support of both my family and the friends that have studied alongside me since starting at university.

Contents

Chapter 1

Introduction

With recent and ongoing developments in high-throughput sequencing methods, there is more genetic data available than ever before. Analyses of this data can provide important insights into genetics as a whole. For example, through analysis of an individual's genome, one could determine their risk for certain diseases to preemptively take steps to minimize it. This means that efficient algorithms are required to keep up with the demand for analyses of each individual.

The traditional method used to discover an individual's genetic variation relies upon the step of *variant calling*. Variant calling involves comparing the genome to a reference genome, and inferring the variants based on where they differ. This is generally done using *alignment-based* methods. Alignment-based methods require aligning a genome's reads to a reference genome so that they can be compared correctly. While highly accurate, this alignment process takes a significant amount of time and computational resources to perform, especially when variations are complex. This makes *alignment-free* methods very useful as they can avoid this costly processing step.

Alignment-free methods skip the variant calling step by use of established knowledge about variants, making them considerably faster and less expensive. These methods rely upon identifying what k-mers (strings of k bases) signify a variant's presence based on prior knowledge, to then look for those in sequenced genomes [26]. These k-mers are called

variant *signatures*. Projects such as the 1000 Genomes Project [4] present overviews of genetic variation amongst the population, and can be used to create indexes of such signatures. While alignment-free approaches are much faster than sequence alignment, it has trouble identifying variants with no unique k-mers that signify them, making it less accurate than the alignment-based methods. Therefore, attempts at improving alignment-free accuracy and speed has been developed in recent years, like Malva [7], PanGenie [9] and KAGE [13], all using different techniques.

KAGE in particular uses two novel ideas to handle these variants. The first idea is to model the expected k-mer counts of the population. This allows non-unique k-mers to still be identified as they would change the expected total count. The second idea is to infer one variant based on the presence of another, which involves analyzing what variants usually occur and don't occur at the same time in individuals. KAGE also considers many possible signatures for each variant when creating an index, in order to find the ones most likely to yield correct results. These ideas, combined with an effective implementation, grants KAGE as good or better accuracy than many other alignment-free genotypers, while also being significantly faster. However, this speed comes at the cost of some time consuming preprocessing steps. [13]

The main bulk of preprocessing for KAGE is creating indexes of expected k-mer counts and variant signatures. While these only need to be made once to be used for multiple individuals, they may need to be updated when the reference genome or variant information changes. The preprocessing steps are also required when creating indexes for other species. As such, it would be beneficial to be able to reconstruct them efficiently. Improving the computational performance of these preprocessing steps is the focus of this thesis.

Chapter 2

Thesis Aims

This thesis focuses on KAGE and aims to explore potential performance improvements for 1) counting the expected *k*-mer counts for a population, and 2) determining the most suitable variant signatures. For use with KAGE and general availability, this will be implemented as a Python module. While it is a priority to make the module work as a stand-alone tool that can be used for multiple aligment-free implementations, it will be made with KAGE integration in mind.

KAGE currently has impractically slow solutions for counting *k*-mer frequency and finding variant signatures. As it stands now, these tasks require several hours to complete for the entire human genome. As such, these are the two main problems I will be tackling. For both problems, the goal is to implement an algorithm that provides KAGE with the necessary *k*-mer information in a much shorter time, while still achieving comparable accuracy.

Throughout the thesis I will explore various avenues that may speed up the indexing process, while fully implementing the promising ones. The goal is for the module to be several times faster than the existing solution while not requiring much more memory. Another goal is to make it practical, as ease of use should make it easier to integrate into existing solutions, such as KAGE.

Chapter 3

Background

The background section will cover the necessary information to fully understand the content of the thesis. This includes understanding biology and genotyping terms, the details of the problems, and relevant software development tools.

3.1 Biology

3.1.1 DNA

Deoxyribonucleic acid, or DNA for short, serves as the language in which genetic information is written. The DNA itself takes the form of two strands forming a double helix where hydrogen bindings between four nucleotide bases encode the data. These four nucleotide bases consist of two purine bases, adenine (A) and guanine (G), and two pyrimidine bases, cytosine (C) and thymine (T). The purine bases are always bound to the pyrimidine bases, in bindings between A and T, and C and G [27, p15]. Thus we can say that the DNA is written in the alphabet $\{A, C, T, G\}$.

Further, if one strand contains the DNA sequence ACTG, the complementary strand would have TGAC at the same location, as A pairs with T and C pairs with G. This is the sequence's *complement*. The *reverse* of this complement is often used for bioinformatics, which would be CAGT. This is then the *reverse complement* of the original ACTG sequence.

3.1.2 Genome

The genome refers to an individual organism's "recipe", with its ingredients, *genes*, encoded in the DNA. While the genome includes all genetic information of an organism, it is often narrowed down to the entire DNA sequence of one set of chromosomes [27, p13]. Humans have 46 chromosomes total, consisting of one pair of sex chromosomes and 22 additional pairs, where each pair consists of one chromosome from each parent [29].

3.1.3 Reference Genome

A reference genome is a representation of a species' genome used to spot irregularities in the DNA sequence of individual organisms. These reference genomes are constructed from multiple individuals in an attempt to make a general sequence that best represents the genetic diversity in the population [15].

The human reference genome is continually getting more refined, with the latest version as of 2022 being GRCh38.p14 [14]. This reference genome was released by the Genome Reference Consortium (GRC), stemming from the Human Genome Project [23].

3.1.4 Alleles & Variants

An allele is one of multiple possible DNA sequences at a specific location in the genome where variation exists. For each such location, an individiual has a pair of alleles, with one allele from each chromosome in a pair. If both alleles in a pair are the same, they are *homozygous*. Otherwise, they are *heterozygous*.

Alleles of a genome that differ from the reference genome are referred to as *variants*. Variants are divided into one of three categories: insertions, deletions and substitutions. Insertions and deletions are collectively called *indels*. For insertions, new bases are inserted into the reference genome. With ACTG as a reference, an insertion of A between C and T would yield ACATG. Deletions on the other hand deletes bases from the reference genome. With ACTG as reference, a deletion of C would yield ATG. Lastly, substitutions both add and remove bases by replacing bases

from the reference genome with other bases. With ACTG as reference, a substitution of T to A would yield ACAG. Further, if a variant is a substitution of exactly one base for exactly one other base, it is called an *SNP (Single Nucleotide Polymorphism)*. If a variant is particularly large and changes many bases, it is called a *structural variant*.

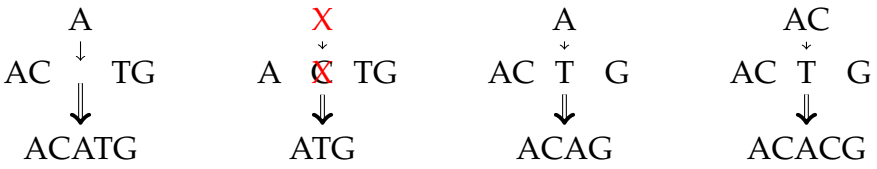


Figure 3.1: *From left to right, examples of an insertion, a deletion and two subtitutions. The first substitution is an SNP while the other is not.*

3.1.5 DNA Sequencing

In order to analyze an organism's genome, its DNA first needs to be sequenced into strings of bases. High-throughput sequencing methods, also called next-generation sequencing (NGS), presents highly efficient methods of sequencing DNA that yield massive amounts of data by way of parallel *short-read sequencing*. [20]

Short-read sequencing involves sequencing many shorter strings of at most a few hundred bases in parallel with thorough quality control. This allows for both accurate and fast sequencing of DNA. These sequences can then be mapped to a reference genome for comparison. However, the shorter reads become weakness when dealing with larger variations such as structural variants. These can make it difficult to map the sequence to a reference genome or to construct whole-genome sequences. [2]

Third-generation sequencing methods address this issue by use of *long-read sequencing*, supporting reads of several thousand bases at once. While this generally has a higher error rate than the short-read alternative, they allow for proper reading of complicated areas of the genome. Short-read and long-read sequencing methods can together provide large quantities of accurate genetic data. [2][17]

3.2 Variant Discovery

3.2.1 Variant Calling

Variant calling is the process of analyzing sequenced data to identify information about variants. After a whole genome has been sequenced, its sequences are *aligned* to a reference genome (see section 3.2.3). The aligned sequences can then be used to find differences from the reference genome to infer what variants an individual has. [3]

3.2.2 Genotyping

Genotyping is the process of determining an individual's *genotypes*. A genotype can have different definitions for different purposes, but within the domain of variant discovery refers to the presence or absence of a variant in an individual's genome for both alleles at a given location. As with pairs of alleles, a genotype is either homozygous or heterozygous based on whether the two alleles are the same or not. When determining an individual's genotype for a specific variant, they have one of three results. The first is to be homozygous for the variant, meaning it is present in both alleles. The second is being homozygous for the reference, meaning it is present in neither. Finally, if the variant is present in one allele, but not both, they are heterozygous for the variant.

3.2.3 Alignment-based methods

Genotyping and variant calling has traditionally been done with alignment-based methods. These methods rely on aligning an individual's sequenced DNA to the reference genome, which is a costly process both in terms of time and memory [26]. Alignment-based genotypers such as GATK [11] show very high accuracy for genotyping at the cost of several hour runtimes and tens of gigabytes of memory. These methods are very important as they allow finding previously unknown variations, compared to the alignment-free counterparts.

3.2.4 Alignment-free methods

Alignment-free methods attempt to resolve the runtime issues of alignment-based methods, and also sometimes lowering the memory requirement. This is achieved by avoiding the variant calling step altogether through the use of established variant information, also avoiding the alignment step in the process [26]. These alignment-free methods are generally less accurate than their counterpart, as they are unable to find unknown variations, but they make up for this in terms of performance. Solutions such as KAGE can genotype an individual in a few minutes compared to the hours an alignment-based method would require, and also needs much less memory. Instead of alignment, alignment-free methods rely on *k-mers* to identify variants in a genome. *K*-mers are polymers, strings of bases, of *k* length. For example, ACGT would be a 4-mer. The overall idea behind alignment-free approaches is to find rare *k*-mers that usually only exist in a genome if a variant is present [26]. These *k*-mers are referred to as a variant's *signature*, and can then be looked for in an individual's genome to prove a variant's existence or lack thereof.

For alignment-free solutions to function, they first need a vast amount of established variant information. The more variant data available, the more variants these alignment-free methods can prepare signatures for and identify. From 2008 to 2015, the 1000 Genomes Project gathered large amounts of data on human genetic variation for public use [1]. This data has proved indispensable for alignment-free methods that rely on established knowledge.

Alignment-free Solutions

Lightweight assignment of variant alleles (LAVA) [26] presented a baseline methodology for alignment-free solutions, greatly reducing the processing needed for genotyping known variants. LAVA paved the way for iterative improvements to the alignment-free approach, making them both faster and more accurate, especially for indels. Malva, PanGenie and KAGE are prominent examples of alignment-free solutions that iterate on LAVA's work. They both introduce additional techniques to further refine the quality of their results.

Malva [7] uses k-mers centered on a variant as its signatures, but dynamically chooses to use larger k-mers when necessary to uniquely identify a variant. This improves accuracy compared to LAVA's fixed k-mer size of 32. It also supports multiple signatures for the same variant, to handle variation dense areas where multiple k-mers may signify a variant's existence. While this approach functions well, large k-mers become more expensive to process, especially in dense areas. Malva especially struggles with variants that do not have any unique k-mers to use as signatures.

PanGenie [9] improved upon Malva's approach by use of known population information to infer the likelyhood of a variant based on the existence of other variants. This was done using a Hidden Markov Model (HMM), which grows quadratically in memory usage with how many variant states it has to consider. While PanGenie outperforms both the alignment-based GATK and the alignment-free Malva in terms of speed, it uses significantly more memory than both of them.

KAGE

KAGE is yet another alignment-free solution that builds upon the work of LAVA, Malva and PanGenie, with new ideas to make it both faster, less memory-intensive and more accurate. Much like Malva, KAGE supports multiple signatures for a single variant, but unlike Malva, these signatures are chosen more carefully. Rather than centering the signatures on the variant, KAGE's approach considers several different k-mers that span the variant to find the best signature available. It also has a second set of signatures that instead signify that the reference is present, rather than the variant, to further refine its results. In addition, to handle variants with non-unique signatures, KAGE considers the expected number of occurences of each k-mer in the genome. This way a non-unique signature can be identified if the expected count changes. Both the counting of expected k-mers and evaluation of signatures are preprocessing steps that are done before genotyping an individual, and thus do not contribute to the actual time spent genotyping. This allows KAGE to be significantly faster and more memory-friendly than Malva and PanGenie, while also achieving a higher accuracy.

3.3 The K-mer Indexing Problem

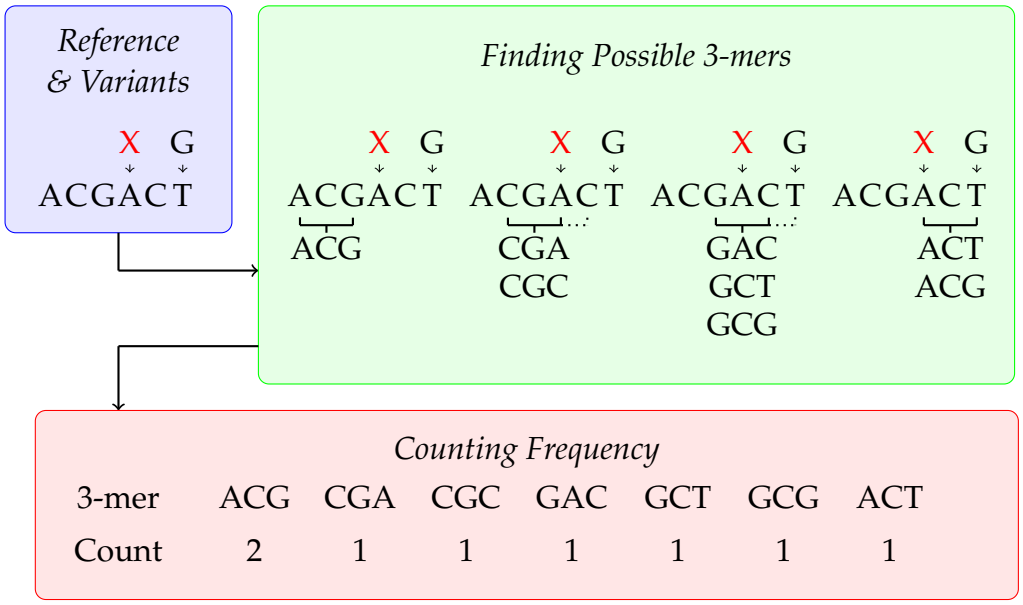


Figure 3.2: *Example of an input sequence with two possible variants and its resulting 3-mer frequencies. In the case of the deletion of A (signified by the red X), the next base in the sequence is appended to the 3-mer.*

Whether it comes to counting the expected occurences of k-mers or determining variant signatures, gathering information about which k-mers exist and what variants and parts of the reference genome they span are of utmost importance. When the genome is dense with possible variants, the many combinations can make this indexing process taxing. As such, fast solutions for this process are of great benefit to genotyping. Figure 3.2 shows an overview of this process for 3-mers in a short sequence with one deletion and one SNP variant.

3.3.1 Existing Solutions

There are a few existing solutions to the k-mer indexing problem, such as vg (variant graphs) [33] and odgi (optimized dynamic graph implementation) [25]. Both of these are C++ implementations with detailed command-line interfaces with many highly sophisticated graph-based operations. This contributes to making them more complex and less accessible compared to a Python module focused on this specific task.

KAGE uses its own implementation for this problem that is provided by the graph-kmer-index module [12]. Contrary to vg and odgi, this is a Python module, but it is also severely slower than its competitors. As such, an optimized Python module for this task would prove useful.

3.4 The Variant Signature Problem

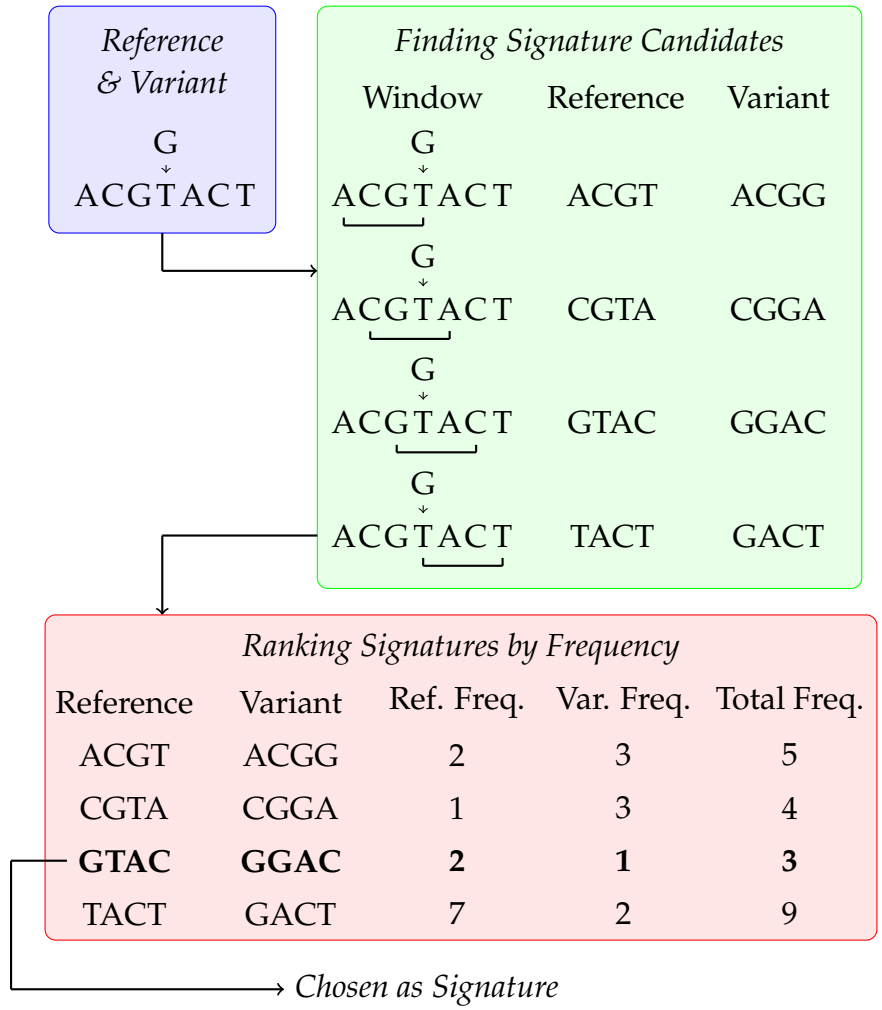


Figure 3.3: *Example of determining 4-mer signatures for a variant. The frequencies used for ranking are from a hypothetical frequency index constructed prior to finding signatures.*

When using a more pre-determined approach for variant signatures such as Malva's where they are simply centered, finding variant signatures is fairly straight forward. However, with KAGE's approach of finding the optimal signatures, this process quickly grows demanding. Several k-mers

have to be considered for each variant, and for each of them their rarity needs to be compared against the expected *k*-mer counts to determine the most suitable signatures. An overview of this process for 4-mer signatures of a single variant is shown in figure 3.3.

3.4.1 Existing Solutions

There are currently no other solutions that determine variant signatures in the same thorough manner that KAGE does, whose implementation is very slow. Therefore, this is one of the central processes in need of an alternative with better performance.

3.5 Data Formats

3.5.1 FASTA & FASTQ

chromosomes.fa

```
>1
acccacttttgatgaggttgtttggtttttccttctaaatttgtttaag
ttccttgtagattctggatattagccctttgtcagatggatagattgcaa
...
>2
aaattttctcccattctgtaggttgcctgttcactctgatgagagtttct
...
```

Figure 3.4: *Example of a FASTA-file's contents. The descriptions denote chromosome numbers.*

FASTA and FASTQ are file formats used to store sequences of single letter codes, and is often how DNA sequences are stored. The structure of FASTA files are quite simple, with a description line starting with a greater-than (”>”), followed by the entire sequence over the next lines. One FASTA file can include multiple sequences, where a new sequence is denoted by another greather-than on the next line following a previous sequence. This allows all chromosomes to be in a single file. [34]

FASTQ files are an extension of FASTA files that also include quality scores for each sequence [10]. These quality scores specify how likely it is

for the sequence to have been sequenced incorrectly [16].

3.5.2 VCF (Variant Call Format)

variants.vcf

```
##fileformat=VCFv4.1
...
##INFO=<ID=AC,Number=A,Type=Integer,Description="Total number of
   alternate alleles in called genotypes">
##INFO=<ID=VT,Number=.,Type=String,Description="indicates what
   type of variant the line represents">
#CHROM POS     ID  REF  ALT  QUAL  FILTER  INFO        FORMAT
21     9411239 .   G    A    100   PASS    AC=1;VT=SNP GT
21     9411245 .   C    A    100   PASS    AC=4;VT=SNP GT
21     9411264 .   A    C    100   PASS    AC=1;VT=SNP GT
...
```

Figure 3.5: *Example of a VCF-file's contents.*
Many more options can precede actual data.

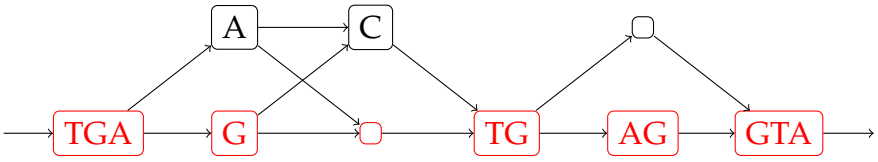
Variant information is stored in the VCF file format. This format allows for the definition of several tags and options used to further describe each variant. The actual variant data is stored in a tab-delimited manner. Each row includes the relevant chromosome number, the variant's position in the reference genome, what bases are changed, and other useful metadata. They can also outline genotype information for one or more specific individuals. [31]

3.5.3 Genome Graphs & GFA

A genome graph is a representation of a reference genome and variants tied to it. Whereas a reference genome acts as a standard to compare other genomes to, a genome graph allows the representation of not only the standard, but also the genetic variation. This makes them useful for genotyping as they also support standard graph traversal approaches to find *k*-mers that help identify variants.

A graph, $G = (V, E)$, of length n_G is defined as a set of vertices (generally refered to as nodes), $V = \{v_0, v_1, ..., v_{n_G-1}\}$, and a set of edges,

$E = \{(x_0, y_0), (x_1, y_1), ...\}$, where x and y are nodes from V. Further, each node, v, of length n_v is defined as a sequence, $S = \{s_0, s_1, ..., s_{n_v-1}\}$, over the alphabet $\Sigma = \{A, C, T, G\}$. Finally, the nodes that originate from the reference genome is a subset of nodes, $R \subseteq V$.



sequence.fa

```
>1
TGAGTGAGGTA
```

variants.vcf

```
#CHROM POS ID REF ALT ...
1      3   .  G   A   ...
1      3   .  G   GC  ...
1      6   .  AG  .   ...
```

Figure 3.6: *Example of a section of a genome graph and its source files. Reference nodes are red while variants are black. Here deletions are represented by empty nodes, but a graph does not necessarily have empty nodes. The edges going to said empty node could instead go directly to the node after.*

Genome graphs can be constructed from pairs of FASTA-files and VCF-files. There is also a format to describe graphs directly, GFA (Graphical Fragment Assembly). The GFA format can describe nodes, the edges between them, and even overlaps in more complicated files. Nodes are specified with S, while edges (called links) are with L. Paths can also be specified with P, for example to list what nodes are reference nodes. [30]

The main caveat of using GFA-files is the loss of variant information only the VCF-file can provide. A VCF-file can contain much more metadata about a variant than a GFA-file. Examples of such metadata include information about a variant's frequency or the genome it was sequenced from. Furthermore, while the GFA format can distinguish reference nodes from variant nodes by use of a path, the variant can have no node at all if it is a deletion. In such a case, the variant may simply be an edge skipping part of the reference, making it difficult to locate later. A similar issue with locating specific variants occur in dense regions where many variants may overlap. VCF-files can circumvent these issues as each variant is explicitly mentioned.

graph.gfa

```
H     VN:Z:1.1
...
S     299961    TTGAGAGTATCTTCACTGAGTG
S     299962    C
S     299963    T
S     299964    TTAGGTCCATG
...
P     1     0+,1+,2+,...,299961+,299962+,299964+,...
...
L     299961    +     299962    +     *
L     299961    +     299963    +     *
L     299962    +     299964    +     *
...
```

Figure 3.7: *Example of a simple GFA-file where the reference node 299962 has a substitution variant, node 299963.*

3.6 Software Development

3.6.1 Python

Python is a dynamically typed interpreted language, whose interpreter is written in C. It has become a central programming language and is often used for data analysis [28]. However, Python is very slow compared to compiled languages and requires high-performance libraries for demanding operations. As such, Python is often used to quickly and easily glue together complicated tasks from these libraries into more easily readable code. For ease of use, Python features a garbage collector that automatically clears unused memory.

3.6.2 NumPy

NumPy is a Python library for managing large arrays of data and perform efficient transformations [22]. In Python, neither variables nor lists are fixed-type, which make them much less efficient and requires significantly more memory to store. NumPy on the other hand manages and operates

on arrays as fixed-type objects in C instead of Python. Because of this, NumPy is used by almost every Python developer working with large quantities of data.

3.6.3 C and C++

Both C and C++ code is compiled to high performance programs orders of magnitude faster than Python. In addition to their compiled nature, high performance code can also be written thanks to manual memory management. While Python uses a garbage collector to manage memory, C and C++ require the developer to explicitly specify when memory is to be allocated and deallocated. This allows for efficient reuse of the same pieces of memory and can help avoid unneccessary copying of data. C++ is an iteration of the C programming language that introduces object-oriented programming.

3.6.4 Cython

Cython is a library that grants Python access to the performance of C and C++.[6] To achieve this, the Cython code has a more granular syntax compared to regular Python, which then allows the code to be translated by Cython into C or C++ code that can then be compiled. Thanks to hooks created by Cython, this compiled code can the be loaded into Python as a module to utilize high-performance code. Cython code can also act as a wrapper to make the pure C and C++ code accessible for Python. To maximize the performance gain Cython can provide, all major demanding tasks should be done exclusively in C or C++ when possible, returning the result as a Python-object. These characteristics make Cython an excellent tool for writing high-performance code for Python.

Chapter 4

Methods

This chapter will cover the chronological process of exploring implementation options and creating the final Python module. Additional details about performance results, such as datasets, system specifications and the steps to reproduce results can be found in section 8.1 of the appendix.

4.1 Initial Considerations

4.1.1 Python Tools

As the goal was to make a high performance implementation in Python, I quickly decided that I wanted to utilize C. C is a low-level compiled programming language, which means it has very granular instructions and is compiled to machine code. Both of these properties make it suitable for writing high performance code. As such, it was a logical choice to look into Cython (see section 3.6.4) to combine Python and C. With Cython there would be two main options of how to use it. The first is to use it directly, by writing Python-like code, but with some more granular additions in order to properly compile it to C or C++ code with hooks to be Python compatible. The second is for Cython to simply be a wrapper for code written entirely in C or C++, with some intermediate Cython code to connect the two. A mixture of both options would also be an alternative. To properly consider which option would be best, I decided to start exclusively with Cython to gauge the potential of that option.

For Python, NumPy [22] is the go-to module when working with data. Its data structures are a standard amongst Python developers, and as such, are highly suitable as input and output values. I believe NumPy helps the module be both more practical and accessible. NumPy is also fast thanks to its own C implementation, and even has specific support for passing its data to Cython, C and C++.

4.1.2 Graph Representation

An efficient genome graph exploration solution also requires an efficient representation of the graph it will explore. The choice of representation would need to be properly considered and implemented before the graph traversal algorithm could be made.

There are many ways to represent graphs in code, for example...

- Having a list of nodes and a list of edges, similar to the graph definition in section 3.5.3.
- Having a list of nodes and a matrix of edges that spans the entire node count in both dimensions.
- Having a list of nodes and a 2-dimensional list of edges, where each index contains the edges of one node.
- Having a list of nodes, where the nodes are data structures that store their own edges, and possibly other information.

Out of these options, the first two can be immediately excluded. For the first option, such a list of edges would mean that the entire list would need to be searched to find the edges of any one node. This is in $O(n)$ time, compared to the three other options that can do as well as $O(1)$. The second option has issues with very large graphs, as the memory requirement for the matrix scales in $O(n^2)$. From the final two options, the last approach allows passing nodes around as complete units of data, while also making it easier to add more information to each node if necessary. This is contrary to the alternative, where a node's index in the list of nodes would be passed around, not the node itself. Finally, the data structure approach allows for a node's sequence to be intuitively stored

within the structure. Therefore, the last option was chosen.

Going for the final option I decided on two data structures, one for the graph and one for the nodes. The graph structure is quite simple, only having a list of nodes. For the nodes, I decided to start small and only feature the bare minimum: the sequence as a list of characters and a list of edges. A node's ID is determined by its index in the graph's node list, and the lists of edges has node IDs a node has outgoing edges to.

4.2 First Prototype (Cython)

4.2.1 Implementation

The first prototype was made entirely in Cython, to get an overview of the indexing problem itself and how suited a pure Cython solution would be. To start I chose to ignore the variant signature problem and get the actual graph traversal correct first, to then work from there.

To keep the graph's data in C, the graph and node data structures were defined as C structs in Cython. Then, to have the graph itself be accessible as a Python object, I wrapped this struct into a Python class. This class would have the relevant class methods to interact with the data stored in the struct. For this prototype, the graph is constructed from a list of sequences and a 2-dimensional list of edges, similar to that of the third option explained in section 4.1.2. Once the graph has been made, it is immutable as the class has no methods to mutate it.

The methods used to traverse the graph and return its *k*-mers are class-methods for the Graph-class. The `create_kmer_index` function makes a call to the recursive function `get_kmers`. This function then starts at the root node of the graph and explores the entire graph recursively, adding *k*-mers it finds along the way. Each node also has a `visited` property to avoid exploring a path that has already been traversed, and is set to `false` when `create_kmer_index` is called. The found *k*-mers are then added to a regular Python dictionary, where the key is the *k*-mer, and the value is a list of node IDs that *k*-mer was found in.

4.2.2 Testing

Initial testing was done by use of an example program where I wrote input values in the code and manually confirmed if the output values were as expected for a few cases. Once that looked to be in order, I made more formalized tests using a parametrized test with pytest. For this test the correct results were figured out manually for a few different cases including, but not limited to, the tests I did manually in the example program. The test takes the input values and directly compares the output to the manually calculated values. All the tests passed without issues.

4.2.3 Lessons Learned

This prototype mainly served as a proof of concept, with several issues to address during further development. For example it was implemented with a fully recursive method to traverse the graph, which is an approach too naïve for the final implementation. Given the length of genomes and the many genetic variations that may span them, the number of recursive steps would rapidly grow too large. This would in turn likely lead to a stack overflow error, causing the program to crash.

Further, for Cython code to achieve C-like performance, the syntax has to be fairly verbose. Extra syntax has to be added on top of the regular Python syntax, which can grow disorderly when writing large blocks of code. The most obvious example of this is the need to put `cdef` before every variable that will use a C data type. On the other hand it was very practical to be able to access a Python dictionary in the midst of the other high performance code. While such cases work, they are not entirely converted to C, and leave Python artifacts in the code. To ensure that the Cython code is properly converted where necessary, Cython can generate an HTML file with an overview of code that is still an issue. This adds another step to the process when writing high performance Cython code. In addition to this, compiled Cython code made it difficult to determine the source of compilation issues. Instead of the issue originating from a specific line in the original Cython-file, the error would occur a thousand lines down in a C-file generated by Cython. After using Cython for this prototype, I concluded that the better option was to use Cython as a

wrapper for pure C code. This decision was done mainly because it would make for easier troubleshooting, while also allowing for additional tests to be done in C, without the Python-layer on top. However, when interacting with Python objects, some functionality could still be delegated to Cython.

Furthermore, the overall structure of the code got somewhat messy, as it was a first draft. There are two issues I think are particularly important. The first is that while the purpose of the Graph-class is to just represent a genome graph, it has class methods for traversing it. A better structure would leave these methods to another object that reads from an assigned Graph object. The second issue is that the recursive method used to find the *k*-mers is very convoluted. For recursive calls only, a boolean variable is set that makes the function behave quite differently, adding unnecessary noise to the code. The better solution would be to split the initial call and the recursive calls into two separate functions. These issues would be adressed while iterating on the second prototype (see section 4.4).

4.3 Second Prototype (Python)

4.3.1 Considerations

As the Cython prototype had a fair amount of issues, I decided to make a new cleaner prototype in Python that would later be translated to C. While Python is not fast by itself, it is a high-level language with easy to read syntax that can showcase the overall steps of an algorithm. This makes it suitable for prototype development, as one can more easily test new ideas to refine the algorithm itself, before implementing it in a low-level language like C. The goal is to iterate on this prototype until the performance enhancements deemed most important are implemented, before translating it to C. Tests would also be both made and used during development to ensure correct results.

4.3.2 Implementation

The first version of the prototype closely resembled the Cython prototype, but with a change in recursion strategy. Instead of performing recursive calls through the entire graph, the algorithm would iteratively start a

smaller recursive traversal from every node. Each of these searches would terminate when no more bases from the starting node are included in found k-mers. This strategy has four major benefits:

1. There will be no risk of causing a stack overflow.
2. There is no longer a need for each node to have a `visited` flag to ensure they are not explored more than once.
3. Given the predetermined value of k, each traversal from a node would never go further than $k - 1$ bases from the starting node. This makes it possible to use fixed-size arrays or buffers, which becomes relevant when transitioning to them in section 4.4.3.
4. The traversal from each node is a completely separate process, making parallelization much easier to implement.

The one main downside to this approach is that the bases for each node might be handled multiple times by different traversals, instead of just once. However, when exploring multiple paths in a graph, this would be likely to happen anyway. Thus I believed this approach to be best suited for the problem.

Listing 4.1 shows pseudocode for the main method to be called when creating an index, while listing 4.2 is the recursive function for traversal. This implementation returns an index in the same format as the Cython prototype, but this will be changed later to better match the format expected by KAGE.

Listing 4.1: *Pseudocode for the main call that will return the entire index.*

```
# nodes is the graph's list of nodes
function create_kmer_index(nodes, k)
   index ← empty dictionary
   N ← length of nodes
   for node_id ∈ 0 to N
      get_kmers(nodes, k, index, node_id, node_id, "", 0, 0, false)
   end for
   return index
end function
```

Listing 4.2: *Pseudocode for the graph traversal function. The kmer_buffer parameter is the sequence of the current path, and kmer_length is its length. The rec_kmer_length parameter is the kmer_length at the first recursive step, while recursive is boolean and indicates if the current call is recursive.*

```
function get_kmers(nodes, k, index, start_node_id, node_id,
    kmer_buffer, kmer_length, rec_kmer_length, recursive)
   node ← nodes[node_id]
   for base ∈ node.sequence
      # Append each base of the node to the buffer
      kmer_buffer ← kmer_buffer + base
      kmer_length ← kmer_length + 1
      if kmer_length ≥ k then
         # Only add k-mer if buffer has at least k bases
         kmer ← kmer_buffer[(kmer_length - k) to kmer_length]
         add kmer to index for start_node_id
      end if
      if recursive and kmer_length ≥ rec_kmer_length + k - 1 then
         break # Stop if more than k - 1 bases from start node
      end if
   end for
   if not recursive then
      # Cut the buffer to the last k - 1 bases for initial call
      new_buffer_start ← max(kmer_length - k + 1, 0)
      kmer_buffer ← kmer_buffer[new_buffer_start to kmer_length]
      kmer_length ← min(kmer_length, k - 1)
   end if
   if not recursive or kmer_length < rec_kmer_length + k - 1 then
      # Recurse further if not in recursion or not k - 1 bases away
      for edge ∈ node.edges
         new_rec_kmer_length ← kmer_length
         if recursive then new_rec_kmer_length ← rec_kmer_length
         get_kmers(nodes, k, index, start_node_id, edge,
             kmer_buffer, kmer_length, rec_kmer_length, true)
      end for
   end if
end function
```

4.4 Iterating on the Second Prototype

4.4.1 Changing the Output Format

The current output format of the prototype did not adhere to the output KAGE would expect. The format expected is two NumPy arrays, one of which has *k*-mers and one that has the nodes each *k*-mer was found at. Figure 4.1 shows the difference between the two formats.

Old Output Format	
key	value
ACT	[0, 3]
CTG	[1]
TGA	[2, 3]

New Output Format	
kmers	nodes
ACT	0
CTG	1
TGA	2
TGA	3
ACT	3

Figure 4.1: *Example of output format difference with 3-mers.*

In addition to this, KAGE expects the *k*-mers to be 2-bit encoded, the implementation of which is explained in the next section.

4.4.2 2-Bit Encoding Output

As there are four possible bases (A, C, T and G), two bits are enough to represent one base. One example would be, $A = 0_{10} = 00_2$, $C = 1_{10} = 01_2$, $T = 2_{10} = 10_2$ and $G = 3_{10} = 11_2$, but they may be encoded in any order. Using 64-bit integers, one integer can then represent a *k*-mer up to 32 bases long. This *k*-mer representation reduces the memory required to store bases by up to 75% compared to the ASCII format where each base would require 8 bits each. In practice however, less than 75% is saved because not every 64-bit integer will always store 32 bases.

The goal was to have the returned array of *k*-mers (as in figure 4.1) be 2-bit encoded. This way, the output can be an array of 64-bit integers, instead of a 2-dimensional array of character sequences.

As bases will be encoded billions of times for just a single run through a human genome graph, an efficient encoding implementation

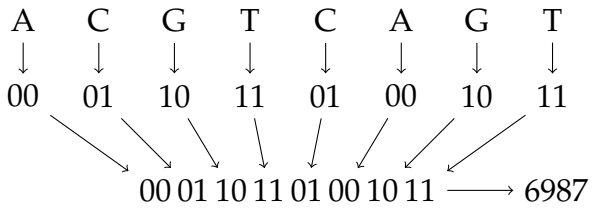


Figure 4.2: *Example of encoding an 8-mer with the encoding $A = 0$, $C = 1$, $G = 2$ and $T = 3$.*

is necessary. More specifically, it needs to be in O(1) time per base, or O(k) time per k-mer. I thought of two different solutions to this problem.

The first solution was to use a map. That would be an array where the ASCII value of each base would serve as an index, and that index has the correct encoded value. For example `index['A'] = 0` and `index['G'] = 2` for an encoding where $A = 0$ and $G = 2$.

The second solution was an entirely bit-operation based approach. After investigating the binary values for the four relevant ASCII characters, I discovered that the encoding of $A = 0$, $C = 1$, $T = 2$ and $G = 3$ would support such an approach. They supported this idea as the two second-to-last bits of the ASCII values corresponded with the desired encoded values (as shown in table 4.1). This meant that it was possible to avoid using a map and instead encode the bases with a simple bit-wise operation:
`encoded = (character >> 1) & 3`

Base	Upper-Case Binary	Lower-Case Binary
A	0100 0**00**1	0110 0**00**1
C	0100 0**01**1	0110 0**01**1
T	0101 0**10**0	0111 0**10**0
G	0100 0**11**1	0110 0**11**1

Table 4.1: *ASCII values of A, C, T and G with the notable two bits in bold.*

The bit-operation approach would not support other encodings, but it was almost 15% faster (as shown in table 4.2). As the encoding process would eventually be implemented in C, I made smaller C programs to compare the performance, even though it would be implemented in Python for the prototype. Luckily this was also the encoding used by

KAGE, so at this point I decided to choose the performance improvement. However, later in development I went back on this decision (see section 4.6.2).

	Map-Based	Bit-Operation
Time	6795ms	6009ms

Table 4.2: *Speed comparison of map and bit-operation approaches encoding 100 million 31-mers. The median time after seven tests.*

4.4.3 Utilizing NumPy

At this point, the prototype still used many high-level Python operations and structures, such as string concatenation. To move the prototype towards code more translatable to C, the buffers where changed to fixed-size NumPy arrays. As NumPy stores data in a C-like format with a Python interface and can perform efficient operations on the data, I hypothesized that this would also improve the performance.

Thanks to the style of recursion discussed in section 4.3.2, I knew the exact size these fixed-size arrays needed to be. This made the move to NumPy arrays much easier overall. By encoding the Python strings to ASCII, the NumPy arrays could also use 8-bit integers.

The prior string concatenation solution would pass the current *k*-mer buffer to each recursive call, and create new strings when concatenating. With NumPy on the other hand, all recursive calls share the same buffer, which is added to when a call starts and cleared when a call ends. This meant that the new solution had to manage a fair bit more array indexing than the previous solution.

	Python Strings	NumPy Array
Time	1800ms	19308ms

Table 4.3: *Time spent finding and encoding all k-mers for dataset 1 (see appendix 8.1.2). The median time after seven tests.*

Contrary to expectations, the new NumPy solution was significantly slower than the Python string solution (as shown in table 4.3). I believed

this to be because of overhead in NumPy's implementation. NumPy is designed to work with large amounts of data, and therefore has specific methods to quickly perform operations along large arrays. In this case however, NumPy had to handle many smaller operations, where I believe the overhead overtook the time saved. To test this new hypthesis, I made a simple Python program to test setting values individually in a Python-list and a NumPy-array. The results in table 4.4 indicate that this is indeed the case. Regardless, I decided to move on with this NumPy solution, as it would make translation to C easier. In C, this will be directly implemented, and therefore suffer no such overhead.

	Python List	NumPy Array
Time	1711ms	4142ms

Table 4.4: *Time spent setting the value of 100 million elements with a standard Python for-loop. The median time after seven tests.*

In an attempt to make the NumPy implementation perform better in Python, I changed all iterative loops over NumPy arrays to use NumPy's array slicing methods. This still uses a fixed-size buffer, but avoiding Python loops allows more of the actual iterative loops to happen within NumPy's low-level code, and makes the code itself more readable. As can be seen in table 4.5, this slicing made the algorithm perform slightly better, as there was less NumPy overhead to handle.

		NumPy Arrays	
	Python Strings	Without Slicing	With Slicing
Time	1800ms	19308ms	17360ms

Table 4.5: *Time spent finding and encoding all k-mers for the dataset 1 (see appendix 8.1.2). The median time after seven tests.*

4.4.4 Loading Larger Graphs

To help test the implementation with larger sets of data, it needed a way to load that data. For this I decided to create a method that could construct my graph format from an obgraph [24]. These graphs are what KAGE uses, and they are constructed from FASTA and VCF files. As such, this

will ensure that the implementation works well with KAGE. This was fairly straight forward as it just needed to move values from one object to the other. Constructing graphs from source files would wait until later, as that will likely be handled in C.

4.4.5 Differentiate Reference Nodes from Variant Nodes

Until now, the prototype could in no way differentiate what nodes pertain to the reference or to variants. To remedy this, I added a boolean `reference` value to each Node-object. The function that constructs a Graph-object also gained an optional parameter for a list of reference node IDs. If this parameter is provided, all nodes with IDs from said list will be labeled as reference nodes, with the rest being variant nodes.

In addition to this, the `create_kmer_index` method gained a new parameter, `max_variant_nodes`. This parameter allows one to limit recursions to only pass through the specified number of variant nodes. This can be particularly useful for dense graphs where exponential growth makes for a tremendous amount of paths, causing the algorithm to work for an unnecessarily long time. Ignoring these variant riddled paths will likely not affect the results significantly, as it is unlikely for most of them to be real-world cases.

4.4.6 Separate Graph Traversal to Another Class

To make for better quality code, I decided it was best to move graph traversal to a class separate from the Graph-class. The Graph-class is only supposed to represent a graph and handle graph-related methods. Traversal of graphs is outside the scope of the Graph-class' responsibilities. This way, graph traversal methods can be changed separately from the graphs themselves, and one graph can be used across multiple traversal methods. For these reasons I made the KmerFinder-class. This class took a graph, the value of k and the maximum variant nodes as constructor parameters, and uses the function `find()` for the same purpose as the prior `create_kmer_index` function.

Another benefit of using a separate class is that it opens the way to use class-variables for certain values instead of passing them for each

recursive call. These include the NumPy *k*-mer buffer, the value of k, the maximum variant nodes, and the result arrays of *k*-mers and node IDs. This greatly shortened the list of parameteres required by the `get_kmers` function, and will also push less values to the stack.

4.4.7 2-Bit Encoding the Whole Graph

Before translating to C I chose to implement 2-bit encoding of the entire graph, meaning each node's sequence would be 2-bit encoded during construction. This would not only reduce the memory required to store the graph, but also mean all bases are already encoded when indexing *k*-mers, skipping the encoding step when gathering results. It is also beneficial to do this *before* translating to C, as it requires major restructuring of the graph traversal code. This restructuring would take much longer to do in C than in Python.

Hypothetically, encoding the entire graph would be up to k times faster than encoding individual results. For example, if the algorithm was to find 3-mers in the sequence $ACTGCA$, it would find ACT, CTG, TGC and GCA and encode each of them. In this scenario, the bases not close to the edges (T and G) were encoded three times, or more specifically k times. This means that when finding 31-mers, all bases not by the start of end of the entire graph would be encoded 31 separate times. In comparison, if the graph was encoded as a pre-processing step, all the bases in $ACTGCA$ would be encoded exactly once, with those encoded values then being used in result gathering. For a dense graph with many paths, even more encoding steps are avoided.

This is why I believed that much time could be saved by encoding ahead of time. To further confirm this point, I did a direct comparison between two miniature test programs that collect 2-bit encoded 31-mers from a randomly generated 100 million base string. One test had a string of bases with one byte per base and encoded each 31-mer on demand as it was found. The other test had its bases encoded into a sequence of 64-bit integers, before then finding 31-mers with bit-wise operations between pairs of such integers. The results of which can be seen in table 4.6.

I estimated that the time spent encoding should have been divided by

	Encoding Time	Collection Time
Encode *k*-mers on demand	6742ms	
Encode *k*-mers ahead of time	194ms	180ms

Table 4.6: *Time spent by on-demand encoding compared to ahead of time encoding in milliseconds for* $k = 31$.

k in this linear graph case, and the results further prove this point. We can see that $\frac{6742}{31} = 217.483...$, which is very close to the actual time of 194ms. Further, given how quickly the computer can perform bit-wise operations, the gathering of the *k*-mers after the graph preprocessing is also very quick. Even with the graph encoding and *k*-mer collection time combined, the preprocessing solution is 18x faster. This is only accounting for the process of encoding and collecting. The speed increase is likely even higher if buffer updates and other value management while traversing the graph is considered too, as well as for denser graphs where more time is gained on the pre-processing.

Encoding the graph itself mainly involved changing a node's sequence from an ASCII string to an array of 64-bit integers. I also added a node property for how many bases long its sequence is. Then it was only a matter of encoding its sequence into 64-bit integers (32-mers) to fill the array. However, handling this new node sequence format in the graph traversal function required significant changes.

K-mer Buffers

Previously the *k*-mer buffer was a NumPy array of $k * 2$ length as to store any necessary bases during recursion. As the bases are now condensed into integers, this would no longer work. Instead they were replaced by two 64-bit integers serving as buffers. As one base requires 2 bits, this allows for $128/2 = 64$ bases to be stored, and would support any value of k up to 32. These buffer are the `kmer_buffer` and the `kmer_buffer_ext` ("ext" is short for "extended"). The purpose of these buffers is to allow for both efficient storage of bases and quick construction of result *k*-mers with bit-wise operations. The former is used to store the sequence of the start node, while the latter stores concatenated sequences from the nodes

following the start node for recursive calls. As the graph is already stored in a 2-bit encoded format, and the results are also returned that way, all buffer operations can be done in a few bit-wise operations.

Both buffers are left-aligned by default, meaning that for example the k-mer ACTG, which would be encoded as 8 bits (00011011), would be stored with 56 trailing zeros to the right. However, the `kmer_buffer` is shifted to be right-aligned before recursion, for reasons explained later.

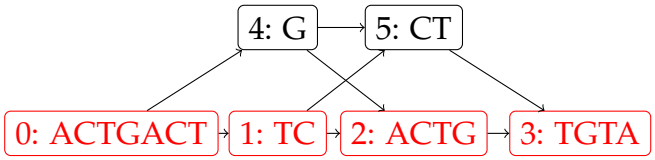


Figure 4.3: *Example graph to showcase how buffers are handled.*

In a graph such as figure 4.3, if a traversal was started from node 0 with $k = 5$, the `kmer_buffer` would simply be set to that full sequence. In cases where the start node has a sequence of more than 32 bases, the first 32 bases will be set first. All k-mers within those 32 bases are saved to the results. Then the first $33 - k$ bases are removed from the buffer before another $33 - k$ bases are appended, and the process is repeated until the entire sequence is explored. This means that for sequences of more than 32 bases, only the last 32 bases are in the buffer when finished. These are the buffer states at this point (the alignment of the encoding is represented by ... on the side of the trailing zeros):

	kmer_buffer	kmer_buffer_ext
bases	ACTGACT	*empty*
encoded	00 01 10 11 00 01 10...	*empty*

When a k-mer is extracted from the buffer to be saved to the results, this formula is used:

```
kmer = (kmer_buffer >> (64 - i * 2)) & mask
```

Here, *mask* is a bit-mask that only keeps the right-most k bases when applied. This mask is premade when the KmerFinder is initialized. i is how many bases from the buffer will be kept, right-aligned, before applying the mask. For this example of 7 bases, ACTGACT, the formula is

done for `i = 7 to k`, adding 3 k-mers total, ACTGA, CTGAC and TGACT. Once all k-mers from the start node are added, the buffer is shortened to only up to the last $k-1$ bases from the start node, as no other bases are needed for recursive calls. The buffer is also right-aligned at this point, for easier bit-wise operations during recursion. Right-aligning the buffer means that the left bit-shifting operations do not need to consider the actual length of this buffer. The buffer states then are as such:

	kmer_buffer	kmer_buffer_ext
bases	GACT	*empty*
encoded	...11 00 01 10	*empty*

Now it is time for the first recursive call, going to node 1. TC is added to the extended buffer, and the k-mers GACTT and ACTTC are added to the results. When the extended buffer is empty, the buffer is simply set to be equal to the node's sequence. The saved k-mers are made with this formula:

```
kmer = (kmer_buffer << i * 2) | (kmer_buffer_ext >> (64 - i * 2)) & mask
```

Here, i determines how many bases will be kept from the extended buffer, and removed from the base buffer. The formula is used for values of i between the length of `kmer_buffer_ext` before and during the current recursion step, excluding the first value. In this case, that means it is done for `i = 1 to 2`, where 0 is excluded (a base from the extended buffer should always be used). The buffers are now as follows:

	kmer_buffer	kmer_buffer_ext
bases	GACT	TC
encoded	...11 00 01 10	10 01...

The recursion now continues to node 2, adding ACTG to the extended buffer. As the buffer is not empty this time, the buffer is updated by:

```
kmer_buffer_ext = kmer_buffer_ext & (full_mask << (64 - i * 2))
kmer_buffer_ext = kmer_buffer_ext | (sequence >> (i * 2))
```

Here, i are the number of bases currently in the extended buffer, *sequence* is the sequence of the current node, and $full\_mask$ is a 64-bit integer with all bits set to 1. The first operation ensures the that all the currently unused

buffer space is set to 0 before the following operation adds the sequence to the extended buffer. After this, the formula shown previously is done for `i = 3 to 4`, adding CTTCA and TTCAC to the results. Note that `i = 5` and `i = 6` are excluded, because those exceed $k - 1$, which means that no bases from the start node are included anymore. For this same reason, the recursion does not continue to node 3. The buffers are as such:

	kmer_buffer	kmer_buffer_ext
bases	GACT	TCACTG
encoded	...11 00 01 10	10 01 00 01 10 11...

With the current search reaching its maximum depth, it backtracks to node 1. The content from node 2 is not cleared from the buffer until new data comes to replace it. Instead, the buffer length is updated so the old values are ignored entirely. The previous steps are then repeated for all paths up to $k - 1$ bases long, before moving on to other start nodes.

Splitting the Function in Two

Given that the second buffer was only needed for the recursive calls, keeping the initial call and the recursive calls in the same function grew impractical and disorderly. Because of this, I split the function in two, making `get_kmers` and `get_kmers_recursive`. This way, `get_kmers` only had to consider the first buffer, before passing its state along to `get_kmers_recursive`, which would then call itself moving forward. This helped make the code much more manageable, and likely also faster as both functions got less to consider. The initial call could ignore the extended buffer entirely, while the recursive calls would only need to consider the first 64-bit integer in a node's array of encoded sequences, as the recursive calls never exceed 31 bases in length anyway.

Performance

With this final improvement to the prototype, the runtimes of each major prototype iteration are shown in table 4.7. As can be seen, encoding the Graph beforehand ended up significantly decreasing the time used by the latest version. However, in order to ensure that all bit-operations were performed correctly I converted Python's numeric values to NumPy 64-

bit integers for several steps of every call. For the Python prototype, this added a significant amount of overhead that made it much slower. Even so, the prototype served its purpose in ensuring this approach still functions properly and yields the correct results, which it does.

Prototype Version	Time
Python Strings	1800ms
Numpy Without Slicing	19308ms
Numpy With Slicing	17360ms
Encoded Graph	3883ms

Table 4.7: *Time spent finding and encoding k-mers for the dataset 1 (see appendix 8.1.2). The median time after seven tests.*

4.4.8 Thorough Correctness Tests

I made several tests alongside the implementation of improvements, to ensure that everything worked correctly. These are explained in this singular section for convenience and structure. All the tests were added as parametrized tests in pytest, such as for *k*-mer encoding, where tests were added that confirm the expected results from various *k*-mers.

To thoroughly ensure that the implementation yielded the correct results, I tested it with many graphs aimed at discovering specific issues, as well as comparing results directly against KAGE's existing solution, which is provided by the graph-kmer-index module [12]. The most thorough test was to find all *k*-mers for the entirety of dataset 1 (see appendix 8.1.2) and compare them against graph-kmer-index's results. This was done for several values of k, ranging from 4 to 24, and took a long time to complete and compare. Without a max variant nodes filter, the massive list of results were entirely identical, except for the order of results. However, with a max variant nodes filter the results differed, but generally not by much. This was ultimately considered a non-issue, as a filter will already provide an incomplete set of results anyway.

4.4.9 The Final Algorithm

In the end, the graph traversal algorithm prototype became a highly custom depth-first search that starts traversal from each node in the graph until reaching a maximum depth. This solution considers possible recursion depth and stack-size issues, makes parallelization possible, and also attempts to avoid doing work such as encoding more than necessary.

Step-by-Step Traversal Example

The following is a step-by-step example of how the graph traversal explores the graph from a single node, showcasing both the max variant nodes filter and how the buffer is updated. For a full-graph exploration, this procedure would be performed for every single node in the graph. For this example, the buffer will simply be shown as a string of bases that are not encoded. All saved results are also shown at each step.

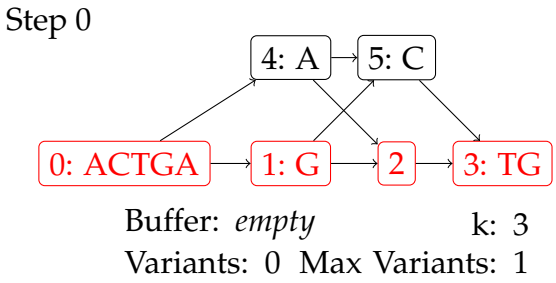


Figure 4.4: *A limited depth-first search is started from the specified node, which in this case is node 0. For this example, k is set to 3 and max variants is set to 1. The path currently being explored will be colored blue, and the results the algorithm writes will be displayed to the right as they are added. The red and black nodes are reference and variant nodes respectively.*

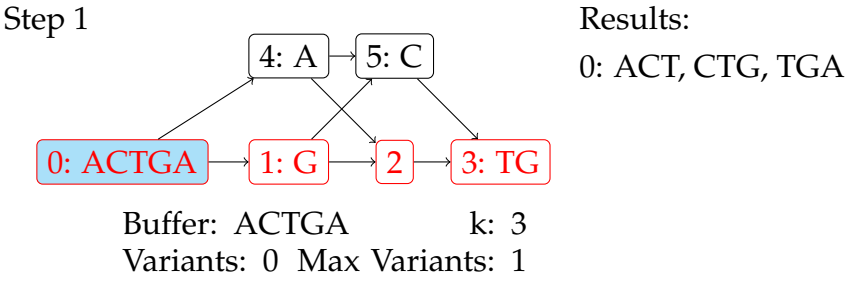


Figure 4.5: *Starting from node 0, the variant counter is not incremented as it is not a variant node. The node's sequence is added to the buffer, which is then iterated through to add all 3-mers, as $k = 3$. Three results are found which are added to the results for node 0. Lastly, the edges of node 0 are explored.*

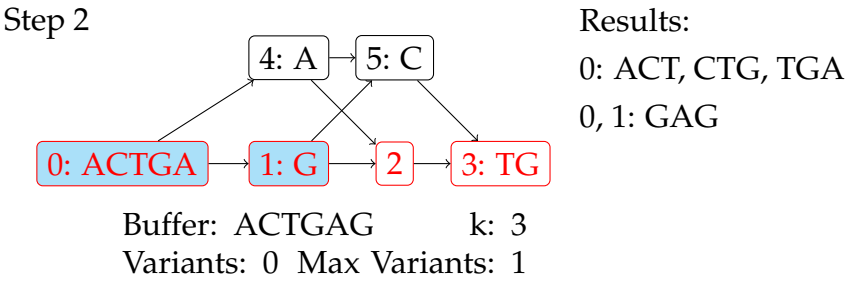


Figure 4.6: *Node 1's sequence, "G", is appended to the buffer. Then all 3-mers that include bases from both nodes in the path are added to the results for those nodes.*

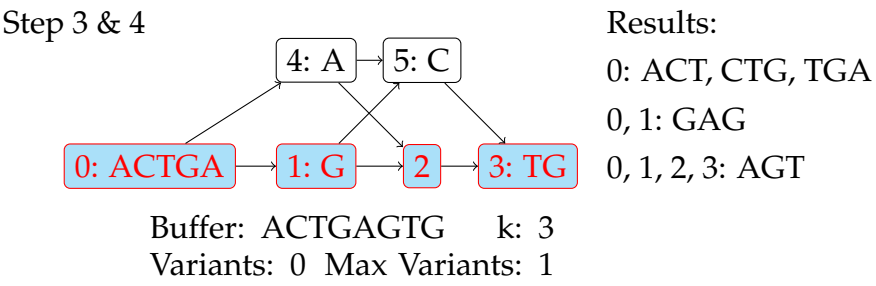


Figure 4.7: *Because node 2 is empty, it is only added to the path before continuing to node 3. Node 3's sequence, "TG", is then appended to the buffer and all 3-mers that span the entire path are added to the results. Note that the 3-mer "GTG" is not added to the results even though it exists in the buffer. This is because "GTG" does not include any bases from node 0. To avoid duplicate k-mers in the results, "GTG" will be added to the results once the starting node is node 1, not node 0. Here the depth-first search terminates and backtracks to node 1's other edge.*

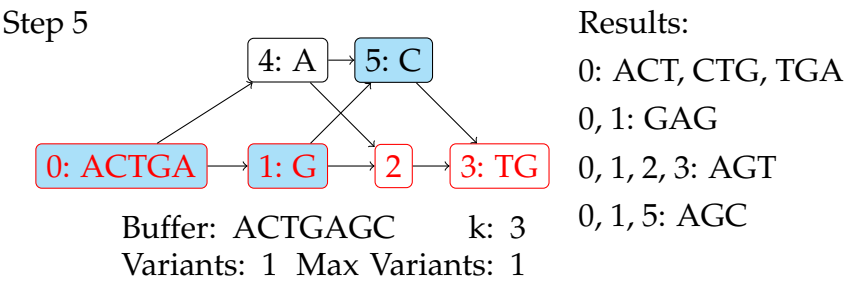
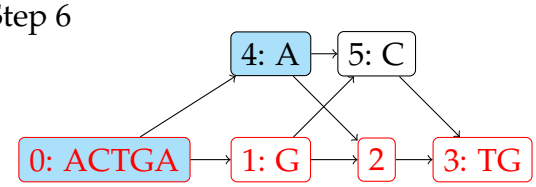


Figure 4.8: *While backtracking, the contents of node 2 and 3 were cleared from the buffer. As node 5 is a variant node, the variant counter is incremented. The variant counter did not exceed the max variants, so the search continues as normal. After "AGC" is added to the results, the search terminates because no bases from node 0 would be included in any new k-mers past this point.*

Step 6

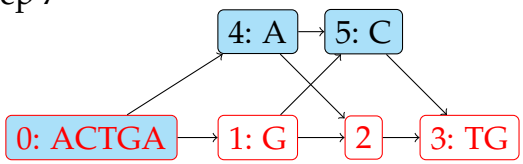


Buffer: ACTGAA k: 3
Variants: 1 Max Variants: 1

Results:
0: ACT, CTG, TGA
0, 1: GAG
0, 1, 2, 3: AGT
0, 1, 5: AGC
0, 4: GAA

Figure 4.9: *After backtracking to node 0, the variant counter is decremented back to 0. When moving to node 4, it is again incremented to 1 and the node is explored as per usual.*

Step 7

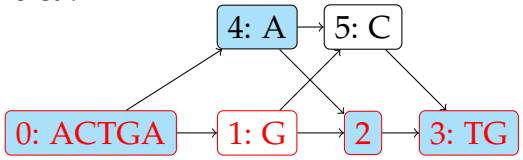


Buffer: ACTGAA k: 3
Variants: 2 Max Variants: 1

Results:
0: ACT, CTG, TGA
0, 1: GAG
0, 1, 2, 3: AGT
0, 1, 5: AGC
0, 4: GAA

Figure 4.10: *Going from node 4 to node 5, the variant counter has now reached 2, exceeding the max variants. Thus, node 5 is not added to the buffer and the search terminates early, backtracking to node 4 to explore its other edge.*

Step 8 & 9



Buffer: ACTGAATG k: 3
Variants: 1 Max Variants: 1

Results:
0: ACT, CTG, TGA
0, 1: GAG
0, 1, 2, 3: AGT
0, 1, 5: AGC
0, 4: GAA
0, 4, 2, 3: AAT

Figure 4.11: *Once again the search skips through node 2 as it is empty before going to node 3. The max variants is not exceeded and new 3-mers are added to the results. After these results are added, all relevant paths from node 0 have been explored. This same search can then be carried out for node 1, then node 2, and so on. Starting from node 1, without a full step-by-step explanation, the 3-mers "GCT" and "GTG" are added to the results.*

4.5 Translating the Prototype to C

4.5.1 Module Setup

Before translating the prototype to C, I decided it would be best to set up the proper Python module development environment. For this I used cookiecutter [5] for a project template. Afterwards I had to configure the project to properly compile Cython files. After ensuring that Cython files for the Graph and KmerFinder classes properly compiled, it was time to translate the prototype.

4.5.2 Considerations

When moving from Python to C, the data types to use for every variable had to be considered. It is important to balance being able to store large enough numbers, while also not using excessively large data types. This is especially true for nodes, as there will be millions of nodes. Thus reducing the size of each individual node will greatly reduce the memory required by the whole graph.

As there are no classes in C, each data structure would need to be represented by a C struct. For the initial C implementation I used data types like `unsigned short`, `unsigned int` and `unsigned long long` to store integers. This would later be changed to more explicit data types like `uint16_t` and `uint64_t` provided by the `stdint` header while improving the implementation. However, for the sake of clarity I will show these explicit data types from the start, as they are more concise and better illustrate the number of bits each integer has.

4.5.3 Graph Representation

The graph struct (shown in listing 4.3) only has an array of `nodes` and the `nodes_len` variable that tells the length of said array. A 32-bit integer was chosen over a 64-bit one for `nodes_len` as it halves the size of each node's edge arrays by limiting the relevant node ID range. This however limits a graph to a maximum of $2^{32} - 1 = 4\,294\,967\,295$ total nodes. Realistically this limit is not expected to be an issue given that each node can have a

sequence of up to 4 294 967 295 bases as well.

Listing 4.3: *Implementation of the graph struct.*

```
struct graph {
   struct node *nodes;
   uint32_t nodes_len;
}
```

The graph's nodes are defined by the following struct,

```
struct node {
   uint32_t length;
   uint64_t *sequences;
   uint32_t sequences_len;
   uint32_t *edges;
   uint8_t edges_len;
   uint8_t reference; // Either 0 or 1
}
```

where each variable is as such:

- *length*: The length of the node's full sequence in number of bases.
- *sequences*: An array of 64-bit integers, each with up to 32 bases.
- *sequences_len*: The length of *sequences*, equal to $\lfloor \frac{length-1}{32} \rfloor + 1$.
- *edges*: An array of node IDs the node has outgoing edges to.
- *edges_len*: Number of outgoing edges.
- *reference*: Whether or not the node pertains to the reference genome.

As mentioned, each node has a maximum length of $2^{32} - 1 = 4\,294\,967\,295$. Thus the *sequences* array can be up to $\lfloor \frac{4\,294\,967\,295-1}{32} \rfloor + 1 = 134\,217\,728$ integers long. The purpose of the *sequences_len* variable is to simplify iteration over *sequences* by removing intermediate calculations. Removing this could be considered if memory usage for graphs become an issue.

The *edges* array holds 32-bit unsigned integers, in accordance to the *nodes_len* variable of the graph struct. Each node supports a maximum of $2^8 - 1 = 255$ outgoing edges, as *edges_len* is an 8-bit unsigned integer.

Much like with node IDs being 32-bit, this was done to reduce the memory required by each node. For genome graphs, it is unlikely that a node will have anywhere close to 255 edges, as that would require an absurd number of variants in a single location.

Lastly, the boolean value *reference* keeps track of whether a node is from a reference or a variant.

For graph construction, I decided to reuse the interface I made for the first prototype. As that prototype was in Cython and already used structs, it was a simple matter to make it populate the new structs.

4.5.4 Graph Traversal

Thanks to the prototype choices, such as using NumPy, the algorithm could more or less be translated to C line by line without issue. The most notable change was to saving results, because I could no longer use Python's dynamic lists. Instead I added a new function, `add_found` that adds these results, and allocates more memory to store them whenever necessary. Any time more space is needed in the result arrays their size is doubled, while a separate variable tracks how many elements are actually in the arrays.

The KmerFinder-class would now be represented by the `kmer_finder` struct, with all the variables previously in the class. These include the graph, result arrays, k, max variant nodes and buffers. Most of the data type choices for this structure were intuitive. Both k and max variant nodes could be 8-bit integers, as they would never get close to 255. Any variables related to *k*-mers would need 64-bit integers and node IDs would need 32-bit integers.

4.5.5 Cython Wrapping

In order to wrap the new C implementation in Cython, Python classes were used to hold the C data structures, with methods where they pass said data structure to the relevant C functions. This made the Python interface close to that of the Python prototype while utilizing C. The wrapper function that required the most handling was the one calling the

`kmer_finder` and gathering the results. After traversing the graph and gathering *k*-mers, the wrapper function had to create NumPy arrays of the correct size and copy the results into them to properly pass them to Python. Thankfully NumPy supports C-like operations within Cython, so I could directly use C's standard memory copy function `memcpy` to move data from the C arrays to the NumPy arrays quickly. Once this was finished, I could free the C arrays from memory as only the NumPy arrays would be needed moving forward.

4.6 Further Improvements

4.6.1 Reversing Results

Not all *k*-mer based applications encode the bases in the same direction, which would make this solution less practical to use if there is a mismatch. To address this, I added a function to *reverse* every *k*-mer in the result arrays, such that they are encoded in the opposite direction (shown in figure 4.12). The *k*-mers are always saved the same way when found, but this function can be called afterwards to reverse the already found results. For the Python interface however, `reverse_kmers` is simply an optional parameter to the pre-existing *k*-mer indexing function, rather than a separate function to be called afterwards. This parameter is `False` by default as it requires additional work for the program.

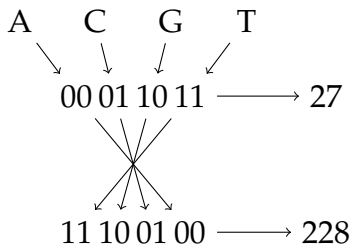


Figure 4.12: *Example of encoding and reversing a 4-mer with the encoding* $A = 0$, $C = 1$, $G = 2$ *and* $T = 3$.

4.6.2 Map-Based Encoding

At this stage of development, I decided to go back on my prior choice of encoding method, instead opting for the map-based approach. While

the bit-operation approach was slightly faster, that performance gain is likely not worth how fragile it is. Supporting multiple encodings will make the module much more accessible in the long run. This was further exemplified when KAGE changed its encoding from ACTG to ACGT during the development of this module. The decoding process of encoded *k*-mers is also simpler with maps.

The map was implemented using an array of 256 8-bit integers as a map. This allows for bases to be encoded and decoded in constant time, O(1). For decoding, the first four indexes of the array, 0 to 3, contain the ASCII value of the base corresponding to that integer. Meanwhile for encoding, both the lower-case and upper-case ASCII value of A, C, T and G serve as indices for the values 0 to 3 (in whichever order they are encoded). Further, any data source with unknown bases saved as N will have these encoded as 0. No other indices of the array are used. An exact example of a map is shown in table 4.8.

Index	0	1	2	3	A	a	C	c	G	g	T	t	N	n
Value	A	C	G	T	0	0	1	1	2	2	3	3	0	0

Table 4.8: *Table of map array indices and values for an encoding of* $A = 0, C = 1, G = 2, T = 3$.

Further, the graph structure was given two new properties. The first was `encoding`, which is a permutation of the string "ACTG" and specifies what encoding the graph uses. The second was `encoding_map` which is the 256 8-bit integer array used for encoding and decoding. Having this map be a property of the graph allows any *k*-mers to be encoded without the need to initialize this map multiple times. The `encoding` variable is only used to create this map, but it is also saved to files (see section 4.6.3) to recreate the map when a file is loaded.

4.6.3 Graph Export and Import

To avoid the need of constructing the same graph every time it is used, functions to save graphs to files and load them later would be useful. This would need an appropriate format to be saved in, which is shown in table 4.9 and 4.10. Other than the actual data and encoding for the

graph, I started the file with a *format code* and a *format version*. Instead of implementing data validation for every step of reading a file, the format code's purpose is to verify that the file being read is of the expected format immediately. While not a foolproof solution, the cases where this validation is passed for a non-graph file would be very rare. The format version, while not necessary during initial development, is meant to future-proof the files. If a new version of the format added more data or restructured it, the version number would allow the program to react to an older format and read it correctly. This old graph would then have any necessary new information added to it, to then be saved in the newest format.

Bytes	Name	Description
10	Format Code	Fixed string of characters. Used to confirm that a file is a graph file.
1	Format Version	Version number between 0 and 255. Used to load old formats correctly.
4	Encoding	The encoding of the graph as a permutation of "ACTG".
4	Node Count	The number of nodes the graph has.
0+	Node Data	Data for each node in order (table 4.10).

Table 4.9: *The properties of saved graphs in the order in which they are saved.*

Bytes	Name	Description
4	Length	Length of the node's sequence.
4	Sequence Length (x)	The number of 64-bit integers used to store the encoded sequence.
$8 * x$	Sequence	All 64-bit integers that make up the encoded sequence.
1	Edges Length (y)	The number of edges the node has.
$4 * y$	Edges	All node IDs the node has edges to.
1	Reference	1 for reference nodes, otherwise 0.

Table 4.10: *The properties of each node in order that are saved with a graph.*

4.6.4 Move to C++

After some deliberation, I concluded that it would be a good idea to move from C to C++. The main motivation behind this was to make hooks to Python more intuitive. Until now, the Python classes for the Graph and KmerFinder stored the C structs and passed these to relevant functions. With C++, the low-level code can also use classes, and those classes have methods. This means the interface in C++ would be more similar to Python than that of C. The Python classes would now only need to store a C++ object of the relevant class and pass calls to Python class functions to the relevant C++ class functions.

With this change, both Graph and KmerFinder were changed from structs to classes, and the methods that previously took them as arguments were made to be class methods instead. However, the nodes remained as structs, as they had no need for class methods.

During the refactoring from C to C++, I also moved from the regular integer data types to those provided by the standard library `stdint`. These data types are more explicit and concise in both their signedness and size. All the types follow the same format where they specify the number of bits (like `int8_t` or `int64_t`) and have the `u` prefix if they are unsigned (like `uint32_t`). Using these data types is also important for compatibility, as the exact size of integer types like `long` may vary from system to system when compiled.

4.6.5 C++ Tests

Thus far, all unit tests were still done by pytest. This meant that the C/C++ code could only be tested through the Cython bindings, which sometimes made finding errors difficult. Therefore I decided to implement some unit tests in C++ in addition to the previous Python tests. For this I used doctest [8], which allows a program to have nested test cases that check correctness for combinations of code. Here I implemented several tests, including tests for all encoding and decoding functions, *k*-mer indexing results, and more later on. These quickly proved useful as they helped me discover a bug in the method that decoded encoded *k*-mers. There

had been no need to decode *k*-mers while developing algorithms, so this method had not been properly tested yet. Valgrind [32] was also used during these tests to ensure the program had no memory-related issues, such as leaks. Running valgrind through the Python interface proved cumbersome, as most reported warnings were related to Python and not the module itself.

4.6.6 Reading GFA

To further make the C++ portion of the implementation independently testable from Python, it would help to implement methods to read one of the file formats for genome data. This would also help make the module more independent from obgraphs that are primarily used by KAGE, thus making it more compatible with other solutions.

The format I chose to first add support for was GFA (see section 3.5.3), as it directly represents the graph to be constructed. The GFA file format supports several more complex notations for things like overlaps. Implementing a parser in C++ that supports every possible syntax a GFA file may contain was outside the scope of this thesis. Instead I focused on GFA files created by vg (see section 3.3.1) from FASTA and VCF files, as they describe a graph as my C++ implementation would.

To implement GFA support in an orderly fashion, I made a C++ class to represent a single GFA file. This class takes the relevant file path and encoding as parameters, and has methods to populate class variables with data from the file. The encoding parameter is used to encode all bases immediately as they are read, which avoids the need to store all the characters of the file at once. Once the entire GFA file has been read, the GFA-class stores all the relevant information in public properties. The Graph-class can then use these values to construct a graph ready for traversal with its `FromGFAFile` method. This helps separate the GFA reading from the Graph-class and makes for more maintainable and well-structured code.

4.7 Finding Variant Signatures

4.7.1 Considerations

When implementing a procedure for finding variant signatures, it was a priority to have it reuse the previously implemented graph traversal algorithm. As that algorithm is already tested to be working, expanding from it would naturally be a good starting point. This approach also avoids the unavoidable repetitive code had this been implemented separately.

While it would be possible to use the node and *k*-mer output of a full graph traversal to determine signatures, I decided against this. Firstly, the full result is incredibly large and uses a ton of memory. Secondly, the entire list of results would have to be searched for the node ID of the variant node and its corresponding reference node for every variant. For these reasons I determined that the better approach would be to handle variants individually by performing a focused traversal around them.

4.7.2 Preparatory Implementation

As they were not needed for the graph traversal previously, nodes held no information about their preceeding nodes. They only had a list of outgoing edges, no ingoing ones. These would be necessary to traverse backwards when making a focused search around a node. Luckily it was a simple matter to add these properties to nodes and populate them during graph construction. With this they were also added as fields to the file format, directly following the outgoing edges.

To properly reuse the graph traversal implementation, I decided to implement flags and filters to modify the behavior of the existing algorithm. All these filters would be handled in a cleanly manner as not to make the algorithm itself messy, but more versatile. This was achieved by only having them modify how each result is handled. That way, the graph traversal algorithm calls a function that adds *k*-mers to results, wherein the flags and filters can modify them as necessary only within that function. For the purposes of variant signature finding, I decided on one necessary

filter, one additional traversal method and one necessary flag.

Node ID Filter

The filter is a node ID filter. This filter makes the algorithm only save *k*-mers for the specified node ID. This allows searches to be performed around a node while avoiding any useless results, and is used to find all *k*-mers a node spans.

Un-Filtered	Filtered (node ID 0)	Filtered (node ID 1)
0: ACT, CTG, TGA	0: ACT, CTG, TGA	*none*
0, 1: GAG	0: GAG	1: GAG
0, 1, 2, 3: AGT	0: AGT	1: AGT
0, 1, 5: AGC	0: AGC	1: AGC
0, 4: GAA	0: GAA	*none*
0, 4, 2, 3: AAT	0: AAT	*none*

Table 4.11: *Example of unfiltered k-mers and their filtered results for node ID 0 and 1.*

Backwards Traversal

In order to find all *k*-mers that span a node, it would not be sufficient to perform a search just from that node. That search would only include *k*-mers starting from a base in said node, not those starting before the node and going into it. As such, it was necessary to make a method for traversing backwards until all *k*-mers spanning the start node were found. To achieve this, I added a return value to the function for adding found results and the graph traversal function. For the result-adding function, this return value represented how many entries were added to the results, which is either 0 or 1 depending on whether or not it was stopped by the node ID filter. The graph traversal further made use of this return value for its own, and returns the total number of results added for the entire traversal from a node. Using this return value, I could traverse the graph backwards and know when to stop traversal. The traversal would stop when a search was started from a prior node without adding a single result that included the start node.

Window-Style Results Flag

Finally, the flag used to find signatures is called *save_windows*, and drastically changes the format of the results. Instead of finding node IDs and k-mers, the algorithm will find all windows of size k spanning a center node. Whenever *save_windows* is used, the node ID filter should be used as well to specify the node to find windows for. Possible windows for a node include all sequences of size k from $k-1$ bases before the node to $k-1$ bases after the node. Two or more sequences share the same window if they include exactly the same bases from the center node. The following is a graph better suited for an example of this functionality.

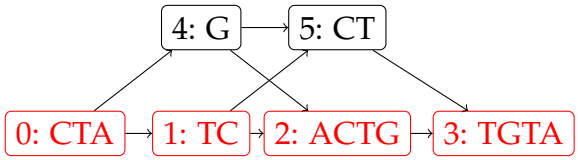


Figure 4.13: *Example graph for window-style results.*

For this example, the node ID filter is set to node 2. As such, all windows for node 2 will be found, and these are the results.

Window Position	k-mers
-2	TCA, AGA
-1	CAC, GAC
0	ACT
1	CTG
2	TGT
3	GTG

The window position is where the sequence starts relative to the first base of the node. Thus, the window at position 0 only has the k-mer ACT, which is the start of the node. Before node 2 there are two possible nodes, 1 and 4. In cases like this, multiple k-mers are saved for the window. For example, at window position -1, CAC and GAC are saved, where they both have the same sequence, AC, from node 2 in common. A window can have many k-mers if there are several possible paths. For instance, if a node has 3 prior nodes, and those nodes have another 2 prior nodes, a window spanning them would have $2 * 3 = 6$ k-mers total. Note that node 5, CT, is not used in any windows, as no k-mer that passes through it can include bases from

node 2.

4.7.3 Determining Signatures

Each variant has two sets of signatures. The first set has *k*-mers that signify the variant being present, while the second has those that signify it is not present (also refered to as reference signatures). Each set of signatures correspond to the *k*-mers for one window saved by the `save_windows` flag for either the reference node or the variant node. The two windows chosen to represent the reference and the variant are required to be aligned. Two windows are aligned if they include the same number of bases to the left or the right of their respective nodes. This definition allows a reference-variant node pair of different lengths to share aligned windows without issues. For example, using 4-mers, the variant at node 6 in figure 4.14 and its respective reference node, node 3, will have the window pairs shown in table 4.12 as signature candidates. As is shown in the table, negative numbers of bases are also considered for alignment.

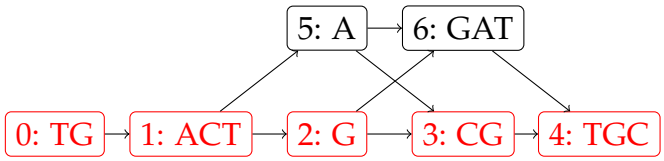


Figure 4.14: *Example graph for variant signatures*

Alignment	Reference 4-mers	Variant 4-mers
3-left	CTGC, CTAC	CTGG, CTAG
2-left	TGCG, TACG	TGGA, TAGA
1-left	GCGT, ACGT	GGAT, AGAT
0-left	CGTG	GATT
-1-left	GTGC	ATTG
-1-right	CTGC, CTAC	TGGA, TAGA
0-right	TGCG, TACG	GGAT, AGAT
1-right	GCGT, ACGT	GATT
2-right	CGTG	ATTG
3-right	GTGC	TTGC

Table 4.12: *The signature candidates for the variant at node 6*

From the signature candidates, the variant signature chosen is the one with the lowest *k-mer frequency*. A candidate's *k*-mer frequency is the frequency sum of the highest frequency reference *k*-mer and variant *k*-mer. For example, for the 3-left pair, if CTGC, CTAC, CTGG and CTAG had frequencies of 2, 5, 3 and 4 respectively, that window's overall *k*-mer frequency would be 9. Essentially, the two *worst k*-mers from the reference and variant *k*-mers are chosen, to consider the worst-case scenario. Then the variant signature is chosen to be the window with the best worst-case.

4.8 Finalizing

4.8.1 Creating an Index

As it was now, the module could not create its own index of *k*-mer frequencies. It could only output the formats shown previously, leaving the index with *k*-mer keys and frequency values to another application. To remedy this, I added the `only_save_initial_nodes` flag. This changes it so that only the starting node ID is saved, as without it, a *k*-mer would end up being counted once for every node it spans.

Further, a method was added to the KmerFinder-class that enables this flag in order to return a frequency index for the whole graph.

Un-Filtered	Filtered
0: ACT, CTG, TGA	0: ACT, CTG, TGA
0, 1: GAG	0: GAG
0, 1, 2, 3: AGT	0: AGT
0, 1, 5: AGC	0: AGC
1, 4: GAA	1: GAA
1, 4, 2, 3: AAT	1: AAT

Table 4.13: *Example of results being filtered by the only_save_initial_nodes flag.*

4.8.2 Reading FASTA and VCF

For the tool to be complete and independent, it needed to be able to read FASTA and VCF files. Much like with the GFA support, both these files were separated into their own classes used to retrieve data from them. To create the graph as efficiently as possible, the program first reads the VCF file and creates every single variant node. It then remembers a sorted list of positions for these variants, so that it can properly cut the reference from the FASTA file into their correct nodes while reading. While adding the reference nodes, edges are added between the relevant nodes. All bases are encoded as they are added to the graph, meaning very few ASCII sequences are in memory at once, lowering the overall memory for this process. Much like the GFA reading implementation, this functionality does not support the full scope of FASTA and VCF files.

4.8.3 Additional Variant Signature Options

To provide further options for variant signature selection, two flags were added to toggle two features. The first flag will cause the selection to to minimize overlaps. A signature window for a variant node and one for a reference node overlap if at least one of their k-mers exist among the other's windows. For example, if the the 5-mer ACTTA is evaluated for being a variant's signature, but that k-mer is also a candidate for the reference's signature, it will not be prioritized. When this flag is active, the window will firstly be the one with the least overlaps, and secondly the one with the lowest frequency. The second flag can toggle whether or not the variant and reference signature windows must be aligned. This was done to allow for measurements of how aligning windows affect accuracy. As aligning windows require more work than not aligning them, alignment is disabled by default even if it was enabled by default prior to introducing this flag. Both of these flags were added as options to the Python-interface through optional parameters to methods.

In addition an option was added to print all found k-mers to standard out instead of storing them in memory. This allows results to be saved to files through C++ rather than Python, which is easier for the user than implementing the writing themselves and is several times faster. For

particularly large graphs, this can also avoid the use of multiple gigabytes of memory by use of disk space instead.

4.8.4 Additional Python Methods

There were still many parts of the C++ implementation that were inaccessible from Python and only used as helper methods behind the scenes when finding *k*-mers or variant signatures. To improve the interface provided, I added access to two new methods. The first was a method to create and return a *k*-mer frequency index, as this was only kept in C++ form until now. This index will then be used for any further variant signature selection, and allows the index to be made with different parameters than the variant selection, such as a different number of maximum variant nodes. The second was a method to find all *k*-mers that span a single specified node, which was previously only used as a helper method when finding the relevant windows for signatures.

4.8.5 KAGE Integration

Once the module was ready to be integrated into KAGE, one of KAGE's developers handled this process. A parameter was added to the KAGE indexer to have it use this module instead of its existing solution. This also allowed the use of KAGE's accuracy tests for both the old and new solutions to compare them. For further tests, I added additional parameters to KAGE's indexer myself that allowed features such as minimizing overlaps and aligning windows to be toggled.

Chapter 5

Results

5.1 The Final Module

The end product of this project was the Python module KIVS that can construct genome graphs and efficiently retrieve *k*-mer information from them. KIVS can be found and installed from its GitHub repository [19]. Specifically, graphs can be constructed from a graph provided by obgraph, from a GFA-file, or a FASTA and a VCF file. Out of these, obgraph is fully functional, while the others have not been implemented extensively enough to accommodate everything they can provide. For retrieving *k*-mer information, the module can quickly find all *k*-mers in the graph or for specific nodes, create frequency indexes of said *k*-mers, and find optimal variant signatures. All these functions have parameters to customize their behavior, such as the max variant nodes filter and reversing encoded *k*-mers for solutions that have them the other way around.

While developed to be a Python module, the C++ source files still provide a complete interface for all the functions included in the Python interface, except construction from obgraph. This exception is due to obgraphs being Python objects, and no solution is implemented for reading the NumPy archives they are saved to through C++. While the C++ interface is not as straight forward to use as its Python alternative, it has more fine-grained methods and makes it possible to use in other C++ solutions that do not use Python.

5.2 Performance

5.2.1 Accuracy

The accuracy tests were performed using KAGE's test functionality for both the full yeast genome and dataset 2 (see appendix 8.1.2) which features a five million base segment of the first chromosome of humans. To thoroughly test accuracy, all combinations of relevant options for KIVS was tested to investigate how each of them compare. For these tests, a KAGE index refers to a k-mer frequency index of only the reference genome, while a KIVS index is one where k-mer frequency is counted from the whole graph, limited by max variant nodes. However, KIVS is also able to construct the reference genome index by having max variant nodes set to zero. Accuracy is divided into two categories: indel accuracy and SNP accuracy. Indels are insertions and deletions while SNPs are substitutions of exactly one base.

As can be seen in table 5.1 and 5.2, the results from KIVS compared to KAGE ranges from better to worse depending on the parameters. Notably, the minimize overlap option greatly increases indel accuracy while window alignment has no notable effect on the better results. The k-mer frequency index also consistently performs better when only counting the reference genome rather than parts of the full graph. Specifically for the yeast dataset, the SNP accuracy is slightly lower for KIVS signatures. It is difficult to pinpoint the exact cause of this, but the difference is not by a large amount (less than 1%).

Frequency Index	Signatures	Minimize Overlap	Align Windows	Indel Accuracy	SNP Accuracy
KAGE	KAGE	X	X	84.2540%	94.6299%
KAGE	KIVS	X	X	84.8577%	93.8504%
KAGE	KIVS		X	81.7147%	93.8202%
KAGE	KIVS	X		85.0951%	93.8392%
KAGE	KIVS			82.0532%	93.8388%
KIVS	KIVS	X	X	83.4711%	93.5844%
KIVS	KIVS		X	72.7848%	93.5539%
KIVS	KIVS	X		83.9050%	93.5789%
KIVS	KIVS			79.9700%	93.5766%

Table 5.1: *Accuracies for the yeast dataset with max variant nodes set to 3.*

Frequency Index	Signatures	Minimize Overlap	Align Windows	Indel Accuracy	SNP Accuracy
KAGE	KAGE	X	X	65.5703%	94.4001%
KAGE	KIVS	X	X	65.9178%	94.4447%
KAGE	KIVS		X	63.8328%	94.5145%
KAGE	KIVS	X		65.7678%	94.5351%
KAGE	KIVS			65.2423%	94.3757%
KIVS	KIVS	X	X	65.2509%	93.5002%
KIVS	KIVS		X	60.1440%	93.5014%
KIVS	KIVS	X		65.1007%	93.5136%
KIVS	KIVS			60.3988%	93.5630%

Table 5.2: *Accuracies for dataset 2 with max variant nodes set to 3.*

5.2.2 Performance

To gauge the performance of KIVS finding all *k*-mers in the graph, I compared its single-threaded runtime and memory usage against the existing solutions provided by vg and odgi. As the main use of KIVS is through Python, the tests were done with that interface. Compared to both vg and odgi that are C++ programs with command-line interfaces, KIVS' runtime and memory usage will include Python overhead. This includes copying C++ arrays into NumPy arrays, effectively doubling the peak memory usage as both arrays are in memory while copying. Tests were done for three different options for KIVS: finding all *k*-mers normally, printing results to stdout rather than store results in memory, and a search that includes results for all nodes a *k*-mer spans.

Implementation	Time	Peak Memory	Results
vg	28.13s	244MB	42 491 910
odgi	13.48s	751MB	42 491 910
KIVS	1.48s	581MB	21 245 955
KIVS (stdout)	4.85s	84MB	21 245 955
KIVS (full)	2.21s	3028MB	125 649 773

Table 5.3: *Runtime and peak memory of finding all 31-mers for the yeast dataset. Median results of 7 test runs.*

As shown in table 5.3, KIVS achieves a much higher speed than both vg and odgi. Both vg and odgi print results to stdout, which makes the KIVS test that also does so the best overall comparison. KIVS is able to find all *k*-mers in the graph about 3x faster than odgi and 6x faster than vg, even while doing a full search with significantly more results. Its memory usage without printing to stdout is also lower than odgi, but about double that of vg. However, this is expected as KIVS' peak memory doubles due to array copying. With printing to stdout, the majority of memory used is to store the graph representation, not any results. This allows KIVS to run on a very low amount of memory. Compared to vg and odgi, KIVS has exactly half the number of results. This is due to vg and odgi considering both DNA strands, meaning each *k*-mer's reverse complement (see section 3.1.1) will also be added to the results, doubling the amount. These reverse complements can easily be calculated from the original results and likely

do not heavily affect the runtime unless they are written to files. When KIVS does write to file, it spends about 3 more seconds than usual. As such, it can be assumed that the runtime would be about 8 seconds total if reverse complements were included, which is still faster than vg and odgi. The memory usage of KIVS would see little to no increase as results are not stored in memory when writing to file.

Further, the performance of determining variant signatures was tested against KAGE's existing solution, with a few different parameters. Both were tested single-threaded with max variant nodes set to 3 and 10, while KAGE was also tested using 16 threads. The results in table 5.4 show that KIVS is significantly faster than KAGE for finding signatures, even when competing against multi-threading. Its runtime also scales much better with higher max variant nodes, as KAGE more than triples in runtime from 3 to 10 variant nodes, while KIVS shows less than a 20% increase.

Implementation	Max Variants	Threads	Time
KAGE	3	16	81.24s
KAGE	3	1	347.61s
KIVS	3	1	43.96s
KAGE	10	16	569.35s
KAGE	10	1	1305.99s
KIVS	10	1	51.83s

Table 5.4: *Runtime of finding all variant signatures for the yeast dataset. Median results of 3 runs.*

5.3 Usage

To provide an overview of how the module is used in practice, in both Python and C++, this section covers how the key functionality is used.

5.3.1 Python

The Python interface is the primary interface for KIVS, and has therefore been optimized the most for ease of use. Once KIVS is imported (see listing 5.1), graphs can be constructed from the various formats with simple

function calls shown in listing 5.2. Afterwards, a KmerFinder can be initialized to use the graph traversal methods as in listing 5.3 and 5.4. The user can then use the returned arrays for their own purposes.

Listing 5.1: *Imports required to use KIVS in Python.*

```
from kivs import Graph, KmerFinder
```

Listing 5.2: *Creating and saving graphs with KIVS in Python.*

```
# From GFA
graph = Graph.from_gfa("file.gfa", encoding="ACGT")
# From Fasta & VCF for chromosome 21
graph = Graph.from_fasta_vcf("file.fa, file.vcf", 21)
# From obgraph provided by KAGE
from obgraph import Graph as OBGraph
obgraph = OBGraph.from_file("file.npz")
graph = Graph.from_obgraph(obgraph, encoding="ACGT")
# Save to file and load from file
graph.to_file("file.kivs")
graph = Graph.from_file("file.kivs")
```

Listing 5.3: *Finding and indexing k-mers with KIVS in Python.*

```
kmer_finder = KmerFinder(graph, 31) # 31-mers
kmers, nodes = kmer_finder.find()
# With optional parameters
kmer_finder.find(include_spanning_nodes=True,
   max_variant_nodes=3, stdout=True)

index = kmer_finder.create_frequency_index(max_variant_nodes=5)
```

Listing 5.4: *Finding signatures with KIVS in Python.*

```
ref_nodes = [1, 5, ..., 503, 523]
var_nodes = [2, 6, ..., 504, 524]
ref_sig, var_sig = kmer_finder.find_variant_signatures(
   ref_nodes, var_nodes)
# With optional parameters
ref_sig, var_sig = kmer_finder.find_variant_signatures(
   ref_nodes, var_nodes, max_variant_nodes=3,
   minimize_overlaps=True, align_windows=True)
```

5.3.2 C++

The C++ interface is mainly intended for use by KIVS and not other users, but it is still fully functional by itself, albeit a bit less intuitive. This is especially due to the lack of the keyword parameters provided by Python. While the C++ interface has several fine-grained operations unavailable to the Python interface, most of these are likely not relevant for most purposes. These include modifying graphs on a per-node basis or manually starting traversals from specific nodes. As such, this section will cover how to achieve approximately the same results as the Python interface did in the previous section.

Listing 5.5: *Required includes to use KIVS in C++.*

```
#include "KmerFinder.hpp" // Includes Graph.hpp
#include "Graph.hpp" // Needed if KmerFinder.hpp is not included
```

Listing 5.6: *Creating and saving graphs with KIVS in C++.*

```
Graph *graph;
// From GFA
graph = Graph::FromGFAFile("file.gfa"); // Default encoding
graph = Graph::FromGFAFileEncoded("file.gfa", "ACTG");
graph->Compress(); // Optimize by merging nodes into longer nodes
// From Fasta & VCF for chromosome 21
graph = Graph::FromFastaVCF("file.fa, file.vcf", 21);
// Save to file and load from file
graph->ToFile("file.kivs");
graph = Graph::FromFile("file.kivs");
```

As shown in listing 5.6, graph construction is very similar to the Python interface. The main difference being that a separate function is used if one wants to specify an encoding. If not specified, the default "ACGT" encoding is used.

Finding k-mers and signatures is where the two interfaces begin to differ. To achieve something similar to Python's keyword parameters, these functions allow flags to be set before they are called. Results are also stored in the KmerFinder-object itself, rather than being a returned value. The `ONLY_SAVE_INITIAL_NODES` flag seen in listing 5.7 is an inversion of

the `include_spanning_nodes` parameter in Python. Finally, the method for finding signatures only accepts one node pair at a time, returning its signature as an object with the relevant data.

Listing 5.7: *Finding and indexing k-mers with KIVS in C++.*

```
// Max variant nodes (3) is specified during initialization
KmerFinder *kf = KmerFinder(graph, 31, 3) # 31-mers
kmers, nodes = kmer_finder->Find();
// With optional parameters
kf->SetFlag(FLAG_TO_STDOUT, true);
kf->SetFlag(FLAG_ONLY_SAVE_INITIAL_NODES, true);
kf->Find(); // Results: kf->found_kmers, kf->found_nodes
// The length of these arrays are equal to kf->found_count

std::unordered_map<uint64_t, uint32_t> index =
    kf->CreateKmerFrequencyIndex();
```

Listing 5.8: *Finding signatures with KIVS in C++.*

```
uint32_t ref_nodes[] = {1, 5, ..., 503, 523};
uint32_t var_nodes[] = {2, 6, ..., 504, 524};
kf->SetFlag(FLAG_MINIMIZE_SIGNATURE_OVERLAP, true);
kf->SetFlag(FLAG_ALIGN_SIGNATURE_WINDOWS, true);
VariantWindow *vw;
for (int i = 0; i < ref_nodes.length; i++) {
   vw = kf->FindVariantSignatures(ref_nodes[i], var_nodes[i]);
   // Signatures are in vw->reference_kmers and vw->variant_kmers
   // Lengths in vw->reference_kmers_len and vw->variant_kmers_len
}
```

Chapter 6

Discussion

6.1 The Effect of KIVS for Genotyping

KIVS could potentially provide great benefits for genotyping or other parts of bioinformatics as it would help quickly handle large amounts of genetic data. This performance increase is especially useful for constructing indexes for more species or for specialized purposes. There may be groups or individuals that do research on specific organisms, but due to fewer computational resources cannot efficiently create the necessary indexes. The availability of KIVS as a Python module combined with performance that allows it to run on consumer hardware benefits these groups and allows them to more easily use solutions like KAGE.

Further, ideas like 2-bit encoding the graph before traversing it proved to greatly improve performance. This knowledge along with how buffers were used to quickly construct k-mers can then be applied to other future solutions to the same problems. Faster implementations open possibilities to make processes like signature selection more thorough to find better signatures, while still performing well. It is also possible to directly source the graph implementation from KIVS and use this for entirely separate algorithms outside the scope of KIVS itself.

6.2 Potential Improvements

There are several potential improvements that could be made to the module, most of which were considered at the start of the project. However, most of these were not necessary for the module to fulfill its intended purpose. Still, they deserve mentioning, as it is likely the solution could become yet faster or easier to use with their inclusion.

6.2.1 Returning All Signature Candidates

During the C++ process that determines and returns the optimal signatures, it first finds every single signature candidate for the variant before selecting one of them. The Python interface has no method that allows this full list to be retrieved, only one that retrieves the final decided signature. There could be cases where this information would be useful for a user of the module to have. However, due to how these candidates are stored in C++, they would require a less intuitive data structure in Python to represent them. One option would be a list of class-objects, where each class-object represents one variant and has its own lists of variant and reference signatures, as well as the relevant frequency information. Compared to a NumPy array, this would be much less intuitive to work with for a user.

6.2.2 Include Reverse Complements

While calculating the reverse complements for the final results is a fairly simple process for a user, it would benefit the module to have an option that provides these results along with the normal results. However, this would also double the memory used to store results.

6.2.3 Command-Line Interface

It would be beneficial for the module to also include a command-line interface. That way, some specific methods from the module could be used without the need to create a Python script. This would help the module be even easier to integrate with existing solutions, especially as existing solutions like vg and odgi both have this functionality.

6.2.4 Parallelization

Parallelization is usually one of the first performance improvements to be made in terms of speed. As the algorithm runs once for every node or once for every variant, performing the search over multiple threads would not experience much of any race conditions or memory writing conflicts. Thus, implementing multi-threading would actually be fairly easy.

However, while the module functions do not implement multi-threading themselves, it is still quite easy to do so. Multiple `KmerFinder` objects can share the same graph, and they only ever read from the graph, never writing to it. In the case of finding variant signatures, this means it would be as simple as splitting the list of variant-reference node pairs amongst multiple `KmerFinders`. The main downsides I can think of for this approach is that it could cause multiple k-mer frequency indexes to be created, and that there is no way to count all graph k-mers in parallel with the Python interface.

Still, the most relevant use-case is not to traverse a single massive graph. This is mainly because the variant signature problem is not limited to one single graph. Instead there are multiple graphs, one for each chromosome. Therefore, the simplest way to parallelize the program is to run one thread for each chromosome and their variants. For this case, no further support for parallelization is required on the module's part.

6.2.5 GPU Processing

It is possible that multi-threading with GPU processing could help the algorithm become even faster than standard CPU parallelization. By utilizing the many processing cores of a GPU, many nodes could potentially be traversed from at once. I found an article about a GPU implementation for breadth-first search (BFS) with promising results that led me to believe it could work for this problem too [21]. However, I did not look far into the implementation of this, as it ended up not being necessary.

6.2.6 32-mer Limit

Due to the 2-bit encoded *k*-mers being stored in 64-bit integers, the implementation is limited to support only up to 32-mers. While it would be possible to support larger *k*-mers by representing them with more than one 64-bit integers, this would require much restructuring of the algorithm. It would also make the results less intuitive to work with, as the *k*-mers would no longer be represented by single variables. If support for *k*-mers longer than 32 bases was to be added, the solution for that would likely be implemented as an entirely seperate algorithm. This alternative algorithm would represent a *k*-mer as an array of 64-bit integers, much like a node's sequence, to support any value for *k*. However, this would make the construction of *k*-mers more complicated, making the process slower, and the results bigger. That is why I believe it would be best to add support for this in a separate algorithm to keep the current 32-mer one as fast and simple as possible.

Chapter 7

Conclusion

Various functionalities and optimizations have been explored throughout this thesis, producing the KIVS Python module. Notable optimizations include thorough use of 2-bit encoding and mindful memory management in C++. Compared to existing solutions for finding *k*-mers, KIVS achieves much higher performance and lower memory requirements, especially when writing results to a file. The variant signature methods also perform much better than their KAGE counterpart, at comparable accuracy. These optimizations help make indexing of genomes available to more people with fewer resources, as well as making it feasable to reconstruct indexes more often. As such, KAGE itself is also much more accessible than before. Thanks to the Python interface, the KIVS module is also more intuitive to use and integrate into more solutions.

Chapter 8

Appendix

8.1 Benchmarking

All unit tests are in the main GitHub repository [19], while all supplementary tests such as performance tests can be found in a separate benchmarking repository [18]. The README of each repository explains how to install the module and how to run each benchmarking test. The benchmarking repository also includes multiple intermediate Python prototypes to compare the performance of each. The units tests that directly compare results against those found by graph_kmer_index are not functional, as the graph_kmer_index module has not been maintained due to not being used, and stopped working as expected late in KIVS' development.

8.1.1 System Specifications

All performance results shown throughout the thesis are done on the same machine. This machine runs on an AMD Ryzen 7 5800X (8-Core, 16-Thread, 3.8/4.7GHz), with 32GB of 3200MHz CL16 DDR4 RAM. Files are read from a WD Black SN750 1TB M.2 SSD, with up to 3470/3000MB read/write speed. The machine used operates as a server running headless Arch Linux, meaning there is no graphical user interface. As such, there are next to no processes on the machine that may significantly impede the performance results.

8.1.2 Datasets

All results use on of four datasets:

- Dataset 1: The first 500 000 bases of chromosome 1 from humans with possible variants. Mainly used during prototype development where a larger dataset would take too long to complete.
- Dataset 2: The first 5 000 000 bases of chromosome 1 from humans with possible variants. Mainly used for testing accuracy against KAGE's solution.
- Yeast dataset: The whole-genome of yeast with possible variants, including all chromosomes. Used for both accuracy and performance tests.

Bibliography

[1] *1000 Genomes Project Web Page*. https://www.internationalgenome.org/. [Accessed 2023-04-27].

[2] Boluwatife A Adewale. "Will long-read sequencing technologies replace short-read sequencing technologies in the next 10 years?" In: *African journal of laboratory medicine* (Nov. 2020). DOI: 10.4102/ajlm.v9i1.1340. URL: https://doi.org/10.4102/ajlm.v9i1.1340.

[3] David Armstrong, Melissa Burke, Laura Emery, Jackie McArthur, Andrew Nightingale, Emily Perry, Sangya Pundir, and Gary Saunders. "Variant identification and analysis". In: *Human genetic variation An introduction* (2022). DOI: 10.6019/TOL.HuGenVar_1-t.2017.00001.1. URL: https://www.ebi.ac.uk/training/online/courses/human-genetic-variation-introduction/variant-identification-and-analysis/.

[4] Ewan Birney and Nicole Soranzo. "The end of the start for population sequencing". In: *Nature* 526.7571 (Oct. 2015), pp. 52–53. ISSN: 1476-4687. DOI: 10.1038/526052a. URL: https://doi.org/10.1038/526052a.

[5] *CookieCutter GitHub*. https://github.com/cookiecutter/cookiecutter. [Accessed 2023-04-21].

[6] *Cython web page*. https://cython.org/. [Accessed 2022-09-03].

[7] Luca Denti, Marco Previtali, Giulia Bernardini, Alexander Schönhuth, and Paola Bonizzoni. "MALVA: Genotyping by Mapping-free ALlele Detection of Known VAriants". In: *iScience* 18 (2019). RECOMB-Seq 2019, pp. 20–27. ISSN: 2589-0042. DOI: https://doi.org/10.1016/j.isci.2019.07.011. URL: https://www.sciencedirect.com/science/article/pii/S2589004219302366.

[8] *doctest GitHub*. https://github.com/doctest/doctest. [Accessed 2023-04-23, updated 2023-03-15].

[9] Jana Ebler, Peter Ebert, Wayne E. Clarke, Tobias Rausch, Peter A. Audano, Torsten Houwaart, Yafei Mao, Jan O. Korbel, Evan E. Eichler, Michael C. Zody, Alexander T. Dilthey, and Tobias Marschall. "Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes". In: *Nature Genetics* 54.4 (Apr. 2022), pp. 518–525. ISSN: 1546-1718. DOI: 10.1038/s41588-022-01043-w. URL: https://doi.org/10.1038/s41588-022-01043-w.

[10] *FASTQ files explained*. https://knowledge.illumina.com/software/general/software-general-reference_material-list/000002211. [Accessed 2022-04-25, updated 2022-04-24].

[11] *gatk Official Web Page*. https://gatk.broadinstitute.org/hc/en-us. [Accessed 2023-04-27].

[12] *graph-kmer-index GitHub*. https://github.com/ivargr/graph_kmer_index. [Accessed 2023-04-27, updated 2023-03-23].

[13] Ivar Grytten, Knut Dagestad Rand, and Geir Kjetil Sandve. "KAGE: Fast alignment-free graph-based genotyping of SNPs and short indels". In: *Genome Biology* (Oct. 2022). DOI: 10.1186/s13059-022-02771-2. URL: https://doi.org/10.1186/s13059-022-02771-2.

[14] *Human Genome Overview Information about the continuing improvement of the human genome*. https://www.ncbi.nlm.nih.gov/grc/human. [Accessed 2023-04-01, updated 2022-02-03].

[15] *Human Genome Reference Sequence*. https://www.genome.gov/genetics-glossary/Human-Genome-Reference-Sequence. [Accessed 2023-04-01, updated 2023-03-24].

[16] *Illumina Measuring sequencing accuracy*. https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/quality-scores.html. [Accessed 2022-04-25].

[17] *Illumina What is long-read sequencing?* https://www.illumina.com/science/technology/next-generation-sequencing/long-read-sequencing.html. [Accessed 2023-04-25].

[18] *kivs benchmarking GitHub*. https://github.com/ZinderAsh/python-kivs-benchmarking/.

[19] *kivs GitHub*. https://github.com/ZinderAsh/python-kivs.

[20] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. "De novo assembly of human genomes with massively parallel short read sequencing". In: *Genome Research* (Feb. 2010). DOI: 10.1101/gr.097261.109. URL: https://doi.org/10.1101/gr.097261.109.

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. "High-Performance and Scalable GPU Graph Traversal". In: *ACM Trans. Parallel Comput.* 1.2 (Feb. 2015). ISSN: 2329-4949. DOI: 10.1145/2717511. URL: https://doi.org/10.1145/2717511.

[22] *NumPy web page*. https://numpy.org/. [Accessed 2022-09-03].

[23] Sergey Nurk et al. "The complete sequence of a human genome". In: *Science* 376.6588 (2022), pp. 44–53. DOI: 10.1126/science.abj6987. eprint: https://www.science.org/doi/pdf/10.1126/science.abj6987. URL: https://www.science.org/doi/abs/10.1126/science.abj6987.

[24] *obgraph GitHub*. https://github.com/ivargr/obgraph. [Accessed 2023-04-23, updated 2023-03-23].

[25] *odgi GitHub*. https://github.com/pangenome/odgi. [Accessed 2023-04-27, updated 2023-04-20].

[26] Ariya Shajii, Deniz Yorukoglu, Yun William Yu, and Bonnie Berger. "Fast genotyping of known SNPs through approximate k-mer matching." In: *Bioinformatics* (Aug. 2016). DOI: 10.1093/bioinformatics/btw460. URL: https://doi.org/10.1093/bioinformatics/btw460.

[27] Gautam B. Singh. *Fundamentals of bioinformatics and computational biology : methods and exercises in MATLAB*. eng. Modeling and Optimization in Science and Technologies, Volume 6. Cham, Switzerland: Springer, 2015 - 2015. ISBN: 9783319114026.

[28] *StackOverflow 2021 Developer Survey*. https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof. [Accessed 2022-09-10].

[29] *Talking Glossary of Genomic and Genetic Terms Chromosome*. https://www.genome.gov/genetics-glossary/Chromosome. [Accessed 2023-05-12, updated 2023-05-11].

[30] *The GFA Format Specification*. https://github.com/GFA-spec/GFA-spec/blob/master/GFA1.md. [Accessed 2023-04-01, updated 2022-06-16].

[31] *The Variant Call Format Specification*. https://samtools.github.io/hts-specs/VCFv4.3.pdf. [Accessed 2022-09-10, updated 2022-08-22].

[32] *Valgrind*. https://valgrind.org/. [Accessed 2023-05-13, updated 2023-04-28].

[33] *vg GitHub*. https://github.com/vgteam/vg. [Accessed 2023-04-27, updated 2023-04-26].

[34] *What is FASTA Format?* http://zhanglab.ccmb.med.umich.edu/FASTA/. [Accessed 2022-09-10].