Reference Graph Construction and Merging for Human DNA

Tom Willy Schiller

Faculty of Industrial Engineering,
Mechanical Engineering and Computer Science
University of Iceland
2016

REFERENCE GRAPH CONSTRUCTION AND MERGING FOR HUMAN DNA

Tom Willy Schiller

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Computer Science

Advisors
Páll Melsted
(imagine some text here)

Faculty Representative
(imagine some text here)

Faculty of Industrial Engineering,
Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, January 2016

Reference Graph Construction and Merging for Human DNA
Reference Graphs for Human DNA
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Computer Science

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
VR-II, Hjarðarhaga 2-6
107 Reykjavík
Iceland

Telephone: +354 525 4000

Bibliographic information:
Tom Willy Schiller, 2016, Reference Graph Construction and Merging for Human DNA, M.Sc. thesis, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, January 2016

"And above all, watch with glittering eyes the whole world around you, because the greatest secrets are always hidden in the most unlikely places. Those who don't believe in magic will never find it."
— Roald Dahl

Abstract

The focus of this thesis lies on the computational challenges faced when reading out human DNA. In particular, reads of the DNA are commonly aligned with the help of a reference string. In this thesis, the opportunities and difficulties of using a graph reference are explored instead.
Several approaches for using graph references have already been proposed, but none of them are used widely in the field. In this thesis the existing approaches are therefore compared and new algorithms are developed which are intended to help spread the usage of graph references further. These algorithms are then collected into the Graph Merging Library GML, which can be used to visualize their behaviour to further the understanding of these methods. In addition to being used for the alignment of reads to reference graphs, genomic graphs can also arise in other situations. The newly implemented algorithms are general enough to also be of use in these scenarios.
Overall, this thesis constitutes a step in the direction of adopting the more complicated but also more powerful graphs rather than the sequential string data within the field of DNA analysis.

Útdráttur

(imagine some text here)

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

bps: The pairwise occurring essential building blocks of DNA are referred to as *basepairs.*

bubble: A *bubble* refers to short alternative paths in a genomic graph, such as A[C,G]T, which refers to both ACT and AGT. We sometimes wish for data formats to enable a simple inclusion of small *bubbles* without the explicit generation of nodes and edges for them, as that would lead to a lot of overhead and decrease human-readability, while the opposite is also often wished for, to simplify algorithmically working with the format (spe, b).

BWT: The *Burrows–Wheeler Transform* is a string compression technique (Bur) which is used during the read alignment. For a good introduction to the usage of the BWT in bioinformatics, see bio, chapter 7.

CIGAR: The *CIGAR string* is a sequence of lengths and operations associated with these lengths that is used when aligning one string to another. Commonly, they contain the operations `M` for direct match/mismatch, `I` for insertions and `D` for deletions. However, many additional operations for skipping bases, clipping, padding and much more have been proposed for various projects (SAM; spe, b,e).

indel: *Insertions* and *deletions* are referred to as *indels.* While edit operations merely locally change a character within a string, an insertion or deletion affects the index of all the following characters, which is why indels usually require more effort to be accurately taken into account.

***k*-mer:** A nucleotide sequence of length k.

pipeline: The *analysis pipeline* refers to any set of programs that are used sequentially to reconstruct the full genetic information of the individual whose DNA was read out based on a vast amount of short reads and a known reference, or to create a list of differences between the individual's DNA and the reference.

read: An individual's DNA is often not read out as one continuous string, but as large amount of short pieces of DNA. These short pieces are referred to as individual *reads*.

run-length encoding: A simple method for encoding sequential data by replacing any runs of several identical elements with an integer counting the amount of identical elements and just one of the elements.

self-index: A *self-index* is a data structure that stores sequential data in a compressed form, enables the retrieval of any part of that data without needing to access the original data at all, and allows efficient pattern searches to be able to quickly perform typical work on the contained data.

snip: A *single nucleotide polymorphisms* is a change of a single nucleotide. Snips are rather straightforward to work with, as the location of the surrounding data is not affected.

variant calling: The process of finding all nucleotide differences between an individual's DNA and a reference. The differences or *variants* are usually reported as positions within the reference, and are often given together with confidence estimates.

Acknowledgments

I would like to thank my tutor, Páll Melsted, for his guidance, support and advice. I also would like to thank Mareva Nardelli, Kristinn Ólafsson and Rosemary Milton for inspiring me to continue working even when facing difficult problems. Finally I would like to thank my family for their ongoing support and curiosity.

Thank you all.

1. Introduction

The aim of this thesis is to help improve current approaches from the field of human genome sequencing, in which the genome of an individual human being is read out using a combination of powerful machines and increasingly complex software algorithms. After the sequencing is complete, a list of the variations between an individual's genetic information and a reference genome can be produced in a process known as variant calling. The list can then be analysed to find previously unknown links between genetic variants and diseases throughout large population groups, but it can also be used within clinical tools, allowing the diagnosis of uncommon diseases.
Many technological advances in recent years have already made it possible to sequence the whole genome of an individual for under 1000 USD, but ideally that price should be reduced even more to simplify further research and to make such diagnosis tools available for more people. At the same time, the quality of these methods should also still be improved, as the process most commonly used inherently generates many errors.

1.1. The Role of the Reference

The process that currently is most often used is based on a machine sequencing several strands of the individual's DNA in many short pieces, called reads. A pipeline of several software tools then aligns these reads to a reference genome, with the aim of either generating the full individual DNA or a list of variations from the reference that has been used.
The reference that the software pipeline uses is often a genomic sequence string. It represents the average DNA of individuals whose DNA has previously been read out, as the DNA of one individual should be similar to the DNA of others.

It is necessary to use such a reference genome, as building up the individual DNA just from the reads themselves without a reference is a highly complex problem which cannot be solved unambiguously due to many repetitive regions within the DNA. Even reconstructing a very short DNA string just from the reads without a reference can lead to drastically wrong results, as can be seen in figure 1.1. One purpose of the reference is therefore to guide the construction of the individual DNA in a meaningful way.

Another purpose of using a genomic reference becomes clear in the case of diagnosing a disease in a patient. It is usually not very helpful to build the patient's full individual

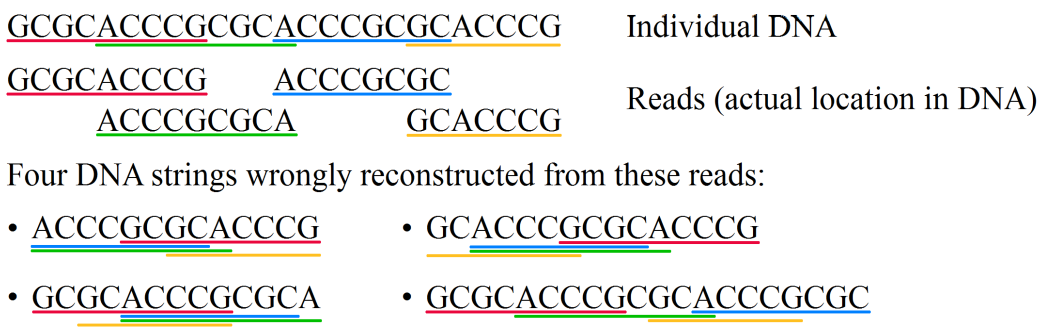

Figure 1.1: Problems with assembling an individual's DNA from small reads without a reference. Four possible reconstructions of the individual's DNA are shown, which all correspond to the observed reads, but are different from the actual DNA of the individual.

DNA, as the plain DNA string without annotations would not give us any information. Instead, we are interested only in the variations from the norm, as these might be known to be associated with a specific disease. By aligning the reads to the reference, the software pipeline can keep track of the encountered differences and in the end give out a list of differences between the individual and the reference. An example for the process leading to such a list of differences can be seen in figure 1.2. The alternative would be to construct

Figure 1.2: General process for generating a list of variations between the individual DNA and the reference.

the entire individual genome, and upon being done with this enormous task to compare the genome with a reference to find all the differences. This comparison however would be a sizeable and difficult alignment problem in its own right, so that it is more sensible to align the reads directly to the reference and give out the differences that were found in that way.

The reference therefore serves both as a blueprint while aligning the reads, as well as an anchor point to which variations within the individual string can be marked.

1.2. The Reference Graph

Traditionally, the reference is a compressed genomic string. In its simplest form it can be thought of as a plain text containing only the letters A, C, G and T.
It is rather straightforward to use such a reference string. Each location within the string can clearly be identified by its position—that is, its distance from the start.
Also, there are very simple file formats available to work with such references. One of these is the FASTA format, which includes the genomic strings together with lines for free text comments and with line breaks which make it easier to display the file in even low level text editors.

The simplicity of reference strings however also has disadvantages. Most importantly, when aligning reads to a reference string, we can only align the reads to the average human genome, instead of aligning them to all known variations of the human genome at once. Not taking the known variations into account can lead to worse alignment results than would be possible otherwise, as the average genomic string may simply not include the particular mutations that have occurred for both the individual whose DNA is supposed to be read out as well as for another individual whose variations from the reference are known.

This is not just a theoretical consideration. Researchers now have access to annotated population variations, which augment the reference sequence string. These annotated variations can be found in databases of common variations, such as the publicly available dbSNP[1] and SNPedia[2]. Even the latest release of the Human Reference Genome, which is designated GRCh38 [3], contains alternative haplotypes for some complex regions, which are essentially known variations of the rigid reference string.
Considering that many alternative references are available and a lot of short variations are known, it becomes clear that the human genome should be thought of as a collection of individual genomes rather than a single rigid reference genome.

Algorithmically, this corresponds to viewing the reference as a graph rather than a linear genome. On this graph, each variation can be represented as a branch which points to the available alternatives. Thereby each personal genome is represented by a path through this variation graph, as can be seen in figure 1.3.
This approach simplifies the identification of common variants, as a variant caller can ideally just note which path has been taken through the reference graph instead of having to explicitly state which differences have been found in which positions.
Using a reference graph also makes it possible to pick up combinations of variants which would currently be lost in the alignment step. Especially the calling of indels, which are usually harder to find and identify than edits, can benefit from having local access to several alternative references which contain already known variants (Alb).

[1] http://www.ncbi.nlm.nih.gov/SNP/
[2] http://www.snpedia.com/
[3] http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/

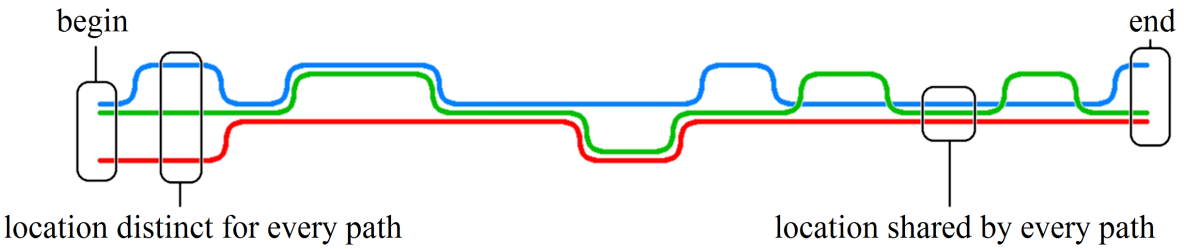

Figure 1.3: Schematic visualization of three paths through a variation graph, each representing an individual's genome.

Another advantage of using a reference graph presents itself when the continuous use of references over time is considered. When a new version of the human reference genome is published, it can take several years for research projects to adapt it, as existing preprocessed, aligned and annotated files all have to be updated (Red).
If instead a reference graph was underlying all alignments, it would be conceivable to update parts of that reference graph without changing the overall indexing of the entire structure, making it possible to seamlessly work with different versions of the entire structure. A possible way for a graph implementation to support this behaviour is by using a designated main reference that is unlikely to change often as the main indexing authority and by adding on top of it variations based on other references which can be changed out without changing the overall index.

Unfortunately, to fully implement a reference graph rather than a linear reference it is necessary to make drastic changes to all of the existing tools in the analysis pipeline. Such changes include the handling of new file formats which are capable of containing graph data, as well as changing the internal algorithmic logic of these tools to work on graphs.

1.3. Thesis Motivation

The focus of this thesis is therefore on the creation of an overview of the existing approaches that can be used for the implementation of genomic graphs, as well as on the exploration of new ways of working with genomic graphs, such as merging compressed graphs implicitly or explicitly.
The goal hereby is not mainly to create a new read alignment tool that uses a reference graph, but rather to create and implement new algorithms that can be used in conjunction with such graphs in the future.

Both the newly created algorithms as well as already established ones for the conversion of graphs between different internal formats will be implemented in such a way that their inner workings are directly observable. This simplifies the future usage of these algorithms within the scope of various parts of the analysis pipeline.

2. Background

Nowadays there are many approaches for aligning short reads to a reference string, and even some work into the alignment of reads to a reference graph has already been undertaken.

2.1. Burrows–Wheeler Transform Without Graphs

The core problem of read alignment to a reference string is simply that a lot of data needs to be worked on, as both the reference itself is very big and the amount of reads is enormous.
This actually leads to two distinct problems, the first being that the memory requirements for storing any of the required data are huge, and the second being that the time that is spend aligning reads for even just a single individual is quite long.

To counteract the first problem, the data that is worked on is usually compressed, which means mostly that the string reference is being compressed in a certain way. A method that has been applied very often is the utilization of the Burrows–Wheeler Transform (Bur), which in its original form consists of reordering a repetitive text with few runs into a text of the same length that contains more runs and can therefore be easily compressed by run-length encoding. A simple example of this behaviour can be seen in table 2.1. As

Table 2.1: Run-length encoding comparison between a highly repetitive string and its BWT. In this example, the run-length encoding of the string itself does not reduce the size, while the run-length encoding of the BWT reduces the size by 29%.

AGCAGCAGCCTTCTTAGCCTT	**Original string**
AGCAGCAG 2C 2TC 2 TAG 2C 2T	**Original string run-length encoded**
TCTCGGGGCTCAAAATTTCCC	**BWT**
TCTC 4 GCTC 4A 3 T 3C	**BWT run-length encoded**

the transformed text contains all the information necessary to recreate the original text, it is not necessary to store the original text at all.
Many current alignment tools such as BWA-MEM (Li2, a) and Bowtie (Lan) use the BWT to compress the reference and even work on the reference in this compressed form directly, without having to recreate the original text in its entirety in memory.

As memory has gotten cheaper over time and machines have become more well-equipped, memory consumption nowadays is not as much of a concern as it once was, when just fitting the reference into memory took a lot of ingenuity.
However, the memory utilization of alignment tools is still important, as further reductions in the overall amount of memory necessary would mean that the alignment could be done using smaller and therefore cheaper machines, which would open up more real-world applications for sequence analysis for which its price currently is still too high. Also, a focus on how the available memory is used could lead to new alignment tools that make more efficient use of the special memory that is more readily available for the processor, such as using the L1, L2 and L3 cache rather than sending a lot of requests to the slower general RAM.
Therefore, research into methods reducing the memory requirements of alignment tools is still ongoing.

2.2. The Role of Graphs in Read Alignment

Initial use of graphs in read alignment was restricted to representing the outcome of the alignment phase, in particular the fully assembled DNA (Mye).

Short reads usually do not make it possible to fully infer the actual structure of the genome that is read out, as different possible structures could have lead to the same reads, especially when considering that there are many errors within the reads that need to be accounted for.
Therefore, often the reads are aligned to a reference and the best fitting position is simply assumed to be the real origin of the read. However, as the decision to choose one position over another can be quite arbitrary, graphs can be used as read alignment output which indicate how the reads are linked together. The true individual DNA that was read out is then one possible path through the graph, while other possible paths through the graph exist that are merely artefacts of the sequencing process.
As such a graph can be quite difficult to work with, not many alignment tools are producing these structures. Instead, the default is usually to just choose the best fitting position and create a read out DNA string. This string will most likely also contain some sequencing artefacts, but will be vastly easier to work with in later steps.

Graphs are currently also starting to be used as a way to reduce the size of several read out human genomes (Li2, b).
The idea stems from the fact that companies or institutions that read out the DNA of several individuals and want to store them can run in problems with the immense file size when storing or transmitting the reads and their aligned locations. Instead, the original read data could be discarded and only the alignment result could be saved and shared—that however would mean that the original data cannot later on be re-interpreted by better algorithms, and can in general not be used any more.
Using a graph in this case can allow several agreeing reads to be combined into long

sequences of unambiguous data, while the locations at which uncertainties exist could still be encoded with all the possible alternatives as different paths through the graph. Therefore, all complicated read data behaviour is contained in the resulting graph, but the size is still vastly reduced when compared to using all reference strings of the population.

2.3. Burrows–Wheeler Transform for Graphs

Graphs have not only been used to indicate the results of the read alignment. The alignment of short reads to a reference graph rather than to a linear reference has also been proposed several times.

A necessary prerequisite for aligning reads to a reference graph is being able to actually create such a graph in the first place.
One of these ways is to take several similar genomic strings and create a graph based on them, such that each of the input strings is a path through the resulting graph. Lee proposed a method to create such a graph from input strings, independent of the order in which these strings are added to the emerging graph.

Storing several very similar references at the same time in a way that not only minimizes the storage space, but also enables efficient pattern searches directly on the stored data was proposed by Mak.
The aim of these techniques is to minimize both the overhead in storage space and the needed time to perform typical work on the references, such as aligning reads to them.

This group of authors continued working with scenarios in which several references are merged into a graph, with reads being directly aligned to the graph rather than to each reference on its own (Sir).
The alignment of reads against the data structure incorporating several references is done in a way that is inspired by the Burrows–Wheeler Transform.

In particular, to be able to use the Burrows–Wheeler Transform for read alignment, a data structure called the suffix array (Pug) is commonly used, which contains information about the locations within the reference. Using the suffix array makes it easier to work on the BWT in its compressed form, rather than having to reconstruct the original string from it to perform work. That is, the BWT together with its suffix array forms a self-index, which is a data structure that contains compressed data and makes it possible to work on that data directly in its compressed form (Nav).
As indexing a graph based on several references is not as straightforward as indexing a single string, which provides an inherent indexing mechanism by simply stating the position within the string, the regular suffix array cannot be used in the case of encoding a graph rather than a string.
However, the plain suffix array is often not used in a practical sense without any changes

anyway. Instead, it is often compressed in some way or another. The reason for that is the immense size of the regular suffix array. A common compression technique is to leave out many of the suffix array's parts and instead compute them dynamically when they are being used rather than storing the entire array in memory.

When using a reference graph for read alignment with the BWT, it is therefore common already to use a structure that simply behaves similar to the suffix array. Such a suffix-array-like data structure can also be constructed for graph references.

In particular, efficient support of the following operations is necessary for such a data structure (Sir):
Given a pattern, the range within the data structure that corresponds to all suffixes of the reference graph that are prefixed with that pattern needs to be found. This essentially provides a functionality for finding arbitrary texts, and is used when aligning reads to the reference, as that is basically a search for the read itself or similar patterns within the reference.
Given a location within the data structure, the corresponding location within the reference needs to be located.
Finally, given a location within the reference, the actual text at that position needs to be extracted.

The main idea is therefore to create a structure capable of fulfilling these requirements, as the remainder of the alignment step is then very similar to the alignment against a reference string.

2.4. Alignment Without an Actual Graph

To avoid the complications arising when working with a complete graph made from several references, other methods have been proposed to achieve similar results without explicitly constructing a reference graph.

One of these is the alignment tool BWBBLE which was created by Hua.
It works on the core assumption that most differences in between two references are snips, which can be encoded with a specific extended alphabet. Then, a modification to the Burrows–Wheeler Transform is made to be able to work with a single string reference containing this extended alphabet.
BWBBLE also supports insertions and deletions which means that it does support more graph-like behaviour, but the core functionality of it is aimed at the extended string reference.

In addition, there have been efforts to more efficiently create the BWT of several strings rather than just concatenating them into a single one. These can help in understanding the challenges of building the BWT of several references (Hol).

2.5. Using Hashes to Encode a Graph

The hashing approach focuses also on a certain way of pre-processing the references to create a certain structure that the reads can then be aligned against.
To use this approach, a specific length k is given, up to which we keep track of sequences in the reference graph.

In the pre-processing step, among all the possible references one main reference is chosen. Then a hash table based on the references is created which assigns each k-mer the positions at which it can be found. Such a position is not as straightforward as a plain integer that gives the index within a string, as the data can lie on branches off the main reference string.
This is solved by using an integer together with minor extra information. The integer points towards a data block, which can be a part of the main reference or a branch of any of the other references that differs from the main sequence. The extra information then points towards the location within the block.

In particular, aligning reads to several references at once in this way has been proposed by Sch.
In this case, the references are taken together to form a graph rather than being used as separate reference strings. For this to be achieved, all references are pre-processed in a special way, with one being taken as the main reference and the other references being stored as changes to this main reference. The pre-processing step results in a data structure that uses k-mers as hashes pointing to the specific location at which these k-mers can be found. Reads can therefore be aligned by finding parts of them in the hash table, looking up the locations that are associated with them, and checking if the entire read can be aligned to that location.
The location here is not just an integer pointing to a character within a string, as it would be for a single reference, but actually a pointer to the particular reference and the location within it. To make this indexing possible, the references are split into blocks that can quickly be addressed with an integer and locations within those blocks.

When the reads are aligned against the data structure, the output of the alignment step can be generated in two ways—either with each read being aligned to the particular reference that it agrees with most, or with each read being aligned in the same way but with the alignment afterwards being rewritten as if the read was aligned to the main reference.
This second option makes it possible to use this approach even as part of an already existing pipeline built for a reference string, not a reference graph, if losing some of the benefits of the graph alignment is acceptable. However, to fully utilize all the graph information, the rest of the pipeline has to be adapted as well.

2.6. Extending the Burrows—Wheeler Transform

Efforts involving changes to and generalizations of the Burrows–Wheeler Transform are also aided by the development of the XBW, which is a method of compressing trees similar to how the BWT is used to compress sequential data (Fer).
Each node in the tree has a label that is one character long. These labels are stored in the XBW together with a special bit vector, which keeps track of whether a given node is a leaf node or not. That is, if a node has a successor, then this bit vector takes the value 0, as that node is not the last node of its branch. If a node does not have a successor, then the bit vector takes the value 1 for that node.

Sir extended the XBW even further, by introducing a second bit vector that enables nodes to have multiple predecessors as well as multiple successors. Therefore, this extended XBW can not only store trees, but instead can keep track of general finite graphs.

3. Methods

To be able to understand the different existing approaches for working with reference graphs and genomic graphs in general, we have re-implemented several of these approaches myself. These implementations have been done in the programming language Python and resulted in several scripts that can be steered by a graphical main program written in Delphi.
We have then focused on creating the Graph Merging Library GML, which is a collection of algorithms written in JavaScript that make it possible to work with reference graphs in various ways. JavaScript is not a particularly fast language in itself, due to many inbuilt functionalities which are helpful for programming in it but slow it down compared to other languages such as C/C++ (Tai). The aim of this library is therefore not to directly provide a means to actually do real-world calculations using the entire human genome as reference graph, but instead it is focused on showing how different algorithms work, and in general to provide a test bed for working with genomic graphs.
Nevertheless, porting these algorithms to a faster programming language and using them in a production environment should of course be possible.

3.1. Data Formats

In the course of working on this thesis we used existing data formats for genomic graphs and designed a new data format as well, to be able to better understand the strengths and weaknesses of the different possible approaches for encoding graphs.

3.1.1. FASTG Format

There are many formats readily available for genomic data strings that can be used when only sequential data is considered. One of the most common formats for genomic strings is the FASTA format. Its simple structure means that it is human-readable and can easily be used by various programs.

However, the aim here is to find data formats that are capable of encoding genomic graphs rather than strings. Such a format is FASTG, which is a FASTA-like format designed to handle graph data (spe, a). Even though encoding genomic graphs is possible in the

FASTG format, we noticed that it has some severe drawbacks.

Files in the FASTG format are unnecessarily big, which is mostly caused by a lot of intentional whitespaces but also by the decision to repeat the default interpretation of each FASTG-specific command just before the command. In the following FASTG example, the underlined text is just repeated and therefore unnecessary information:

...GCATATGTCCTCTCTCC[1:alt|TATGTCCTCTCTCC|GTC]GG...

On the other hand, when eliminating most of the whitespaces to reduce the file size, FASTG files which are already rather difficult to understand quickly become more and more obscure, such that they are no longer easily human-readable. However, that very human-readability is one of the main advantages of FASTA, and without it not much of a reason is left for using this particular family of formats altogether.

The FASTG format also is not very flexible. This can be seen by the fact that it only allows for three layers of variation within the entirety of the data.
The first layer considers global variation that is implemented through a mechanism in which the FASTA comments define which sequences can lead to which others.
The second layer is about local variation that is implemented through FASTG-specific constructs directly in the genomic data.
The third layer represents highly localized variation (such as snips) that can be nested inside of other FASTG-specific constructs from the second layer.
As not all of the constructs can be nested inside of each other and as the comment-based global and construct-based local variation are completely different approaches, this format seems unnecessarily complex.

Finally, the FASTG format as defined in 2012 with its many specialized constructs seems to be excessive for the needs of the Python test pipeline implemented in the course of this thesis.
Trying to implement the entirety of FASTG would in fact also be complicated by the fact that the format is open to amendments, such that solely implementing the existing standard would not be enough to be able to correctly work with all FASTG files which may be encountered. Even the specific bubble notation used within many FASTG files does not actually occur in the standard written in 2012 (spe, a,b).

3.1.2. Bubble Format

For very short and simple human-readable graph data, often the bubble format is used. It is not supposed to be used for production environments and indeed is not complex enough to describe any but the most trivial kinds of graphs. However, we still decided to also have a look into this particular format, as implementing it in a software package is rather straightforward due to its simplicity, and as considering it might teach us valuable lessons about problems to avoid when designing actual graph formats to be used for real

world data.

In the bubble format a genomic sequence is represented as a single continuous string. There are no comments within the format, and neither newline characters nor in fact any kinds of whitespaces.
Any sort of graph representation in STPU is based on bubbles, which can be extended as long as necessary, and can be nested. A bubble is started with a "(" character and ended with a ")" character. Alternatives within the bubbles are marked with "|" characters. A representation of a short and simple graph in bubble notation can be seen in figure 3.1.
Insertions and deletions are represented as bubbles whose alternative route is empty. One such deletion can be seen in figure 3.2.
As remarked before, this format cannot be used to represent arbitrary graphs. An example for which it fails to provide a correct encoding can be seen in figure 3.3. Nevertheless, the bubble format is sufficient for simple algorithmic tests.

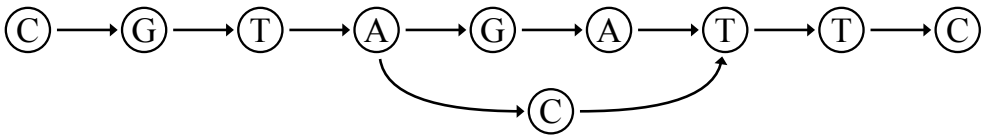

Figure 3.1: Simple bubble, represented in bubble format as CGTA(GA|C)TTC.

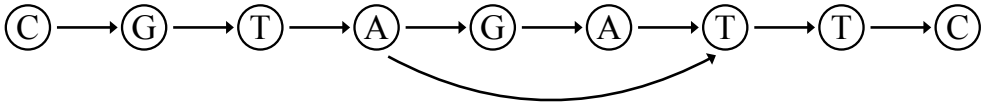

Figure 3.2: Deletion, represented in bubble format as CGTA(GA|)TTC.

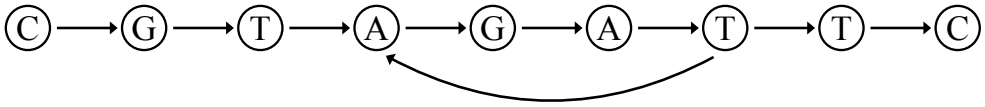

Figure 3.3: Cycle, which cannot be fully represented in bubble format, but can be approximated to any wanted length as CGTAGAT(|AGAT(|AGAT(|AGAT(|...))))TC.

3.1.3. GFA Format

The GFA or Graphical Fragment Assembly format has been proposed in 2014 (spe, b,c). This young format has a lot of advantages over FASTG, mainly that it is a lot more flexible and can easier be read by programs, as not much emphasis is put on making it human-readable. Therefore, the focus can lie clearly on the readability for automated programs.
Unfortunately, GFA has several drawbacks as well.

A crucial problem is that GFA has only shortly been worked on, and no definitive standard has been published.

Although several people have responded quite favourably to the original blog post outlining the format, many have also come up with improvements and outright criticism, making it less likely that the format in its current state is already suited for implementation (kni).
The lack of a unified standard also means that it is not even clear which version of it to base an implementation on.

In the context of this thesis it is also important to consider another drawback. According to Heng Li, the author of the format, GFA only aims to be an assembly format (spe, d). This means that it can represent a graph based on one particular read assembly.
It has not, however, been designed to represent population graphs—which are exactly the reference graphs for which it would be helpful to have a useful format.

Finally, this format is very broad and allows for a lot of complicated graph behaviour, especially considering its elaborate CIGAR strings. This seems to introduce complexity that is, at least for now, unnecessary for the purposes of creating a simple implementation of a graph reference based alignment tool.

3.1.4. GML Format

While programming the Graph Merging Library, we decided to design a new format which could be used to encode genomic graphs. As both the FASTG and the GFA format seem to be rather extensive, the idea behind the GML format is to create a simpler way of encoding graph data. As can be seen with the success of FASTA files for sequential data, the simplicity of a data format is rather important, as it makes it more likely for future tools to be designed with inbuilt support for the format.

The aim of the GML format is also to not unnecessarily complicate the process of making existing tools of the analysis pipeline compatible with graph data. Therefore, the general structure of a GML-formatted file is similar to the general structure of a FASTA file, which is already widely used.
Namely, a GML file can contain comments and data, with different blocks of data being separated by comments. A comment in turn consists of the character “>” to indicate the start of a comment, the name of the following data block, followed by a space character and any free text that can be used as is deemed necessary when the file is created. If a GML file contains no comments at all, then all the rows are simply interpreted as one contiguous data block.

Within a data block, a genomic graph is encoded in two different parts.
The first part is referred to as the “main path.” This is simply the sequence of labels on any one path from the start node to the end node. The start and end nodes are labelled with a hashtag symbol and a dollarsign, respectively. These labels are not included in the main path within the file, as they are implicitly assumed to exist for any such graph.
The second part of a data block is optional. If the second part is given, then the first and

second part of the data block are separated by a pipe character. This second part is an array of info blocks, separated from each other by semicolons.
Finally, each info block consists of exactly four parts which are separated from each other by commas.

- The first part is the identifier of the path, which can contain letters and underscores, as well as numbers in any position but the first. The identifier can also be left empty.

- The second part is the origin of the path, containing the identifier of the path on which this one originates followed by a colon and the position within that path at which it originates.
  The identifier of the main path is `mp`, but in the special case of the main path the identifier and the colon can be left out together, e.g. `mp:8` or just `8` for position eight on the main path, but `path9:8` for position eight on a path with the identifier `path9`. Identifiers need to be defined before they can be used, that is, `AC|a,1,G,2;,a:0,C,3` is valid, while `AC|,a:0,C,3;a,1,G,2` is not valid.
  The counting of the position starts at number zero for the first symbol. However, the main path implicitly contains the hash tag symbol and the dollar sign symbol. Therefore the hash tag symbol on the main path is `mp:0` and the first alphabetical character on the main path is `mp:1`, while the first alphabetical character on a path with the identifier `path9` is `path9:0`.

- The third part is the content of the path, meaning the sequence of labels of nodes on the path. It can be empty if the path consists of just an edge from the origin to the target without containing any nodes.

- The fourth part is the target of the path, specified according to the same format as the origin of the path in the second part of the info block.

The GML format can encode any labelled graphs which start with a special hash tag node and end with a dollar sign node, as long as each node within the graph can be reached from the start and as long as the end can be reached from every node as well. These constraints are acceptable for practical use, as nodes that cannot be reached from the start would be ignored anyway, as well as leaf nodes that are not the end node.
Despite the potential to encode such complicated structures, it is reasonably simple and for very short graphs even human-readable. With increasing graph size the readability of a GML file without special software decreases though, as the info blocks containing alternative paths are quite not located close to where they are found within the main row, but are all concentrated at the end of the file.

3.1.5. FFX Format

In the process of creating GML, we finally designed one more data format for genomic graphs. The idea behind FFX, the Flat Fused XBW format, is to enable saving a flat

XBW table directly, without having to convert it to some other format first.
As shown in subsection 3.14, it can be helpful to fuse several separate graphs together instead of merging them completely. Therefore, FFX is designed to specifically accommodate for these kinds of fused structures as well. It does not however impose these fused structures on the user, as non-fused graphs can simply be stored as single data blocks within an FFX file.

The basic structure of an FFX file is again inspired by FASTA, containing an optional starting comment which contains an identifier for the contents of the file and other text that can be used freely. The optional comment is then followed by one or several data blocks, with each data block being separated by comment lines.
However, if several data blocks are included within one FFX file, they are not seen as separate entities. Instead, the program working on the FFX file is supposed to handle them as fused flat XBW tables, with the end node of the first table leading into the start node of the second table, the end node of the second leading into the start node of the third, and so on.
As more complex formats are less likely to become used by the community at large as it would be harder to re-implement them in various situations, certain rules are imposed on these fused data structures. First of all, a data block is always fused to the one directly following within the file. Theoretically, it would be possible to create a set of rules for the comment lines to indicate that other relations should instead take place, such as the end node of the first table leading to the start node of the fourth and so on, but even though this would increase the flexibility of the format, it would only increase the difficulty of writing an implementation for it. Likewise, each data block is only allowed to contain exactly one hash tag node as start and exactly one dollar sign node as end. Therefore, the path from one data block to the next is always completely linear, and every path through the entire FFX file must use every edge between the data blocks exactly once.
At the first glance, this might seem like a rather limiting constraint. However, it should be noted that much more complex behaviour is very possible within each data block, which can encode a complex a graph as is necessary. Also, this is inspired by the actual needs for a huge graph reference, in which the overall structure is very linear and all perturbations are on a rather local scale.

In FFX, each data block consists of the BWT, the M and F bit vectors, and additional data which could be constructed on the fly from just the BWT, but which takes up a lot of time to be constructed while only take up a small amount of space, such that saving it explicitly within the file means that the lengthy process of computing it again and again does not need to be executed on every opening of the file.

3.2. Data Format Implementations

To compare the different data formats that have been proposed, we wrote several scripts in Python. These scripts supported simplified versions of each step in a complete read

alignment pipeline. The goal here was not to achieve a software package that could compete with the already commonly used alignment solutions, but rather to have a simple test bed for trying out different algorithmic approaches and file formats.

Section 4.1 contains the insights gained from implementing the different data formats.

3.3. Graph Merging Library

After gathering experience on how to implement a pipeline used for aligning reads to a reference in general, we created the Graph Merging Library. This is a software library

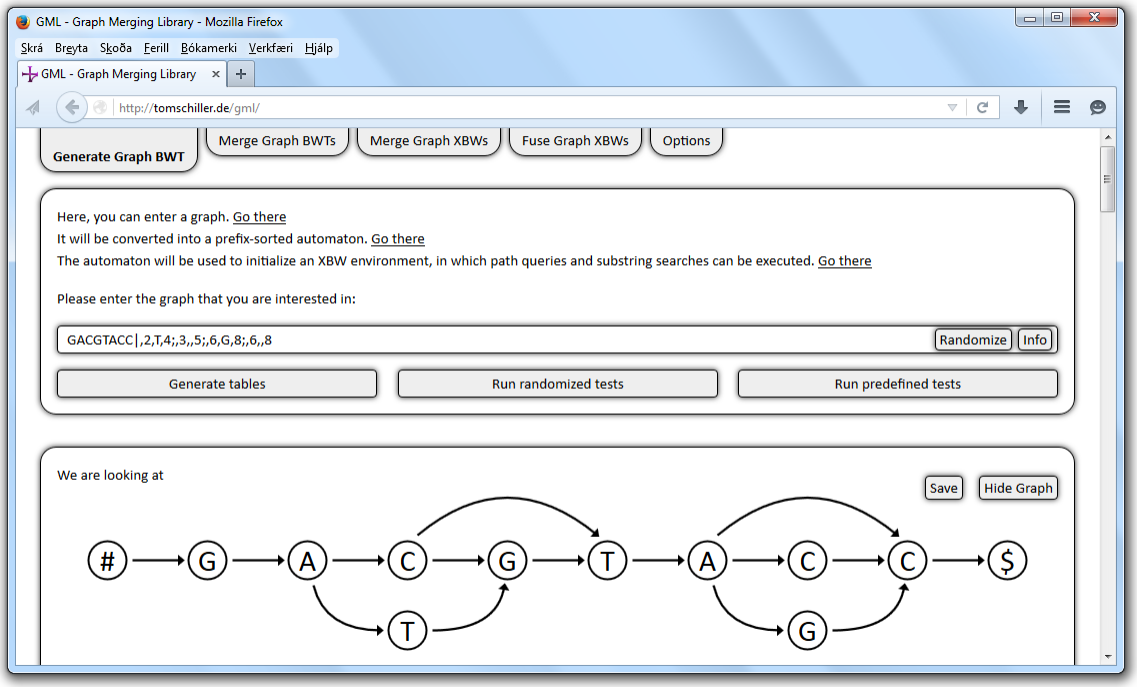

Figure 3.4: Graphical User Interface of the Graph Merging Library.

written in JavaScript, with a graphical user interface written in HTML and CSS, that can be used to more explicitly understand how various algorithms for working with genomic graphs work. A screenshot of user interface can be seen in figure 3.4.
In particular, GML includes different methods for merging and fusing genomic graphs, which may be helpful to other future projects.

3.4. Generating a Genomic Graph

To use a genomic graph in the Graph Merging Library, a GML file can be opened which contains the desired graph. In the graphical user interface, not only can such a GML file be opened, but a valid GML data block can also be entered directly within an input field to simplify getting started with the software. If such a file has been opened, then some work still needs to be undertaken to convert the opened graph into the extended XBW format.
The extended XBW format requires for each graph for start with a node labelled with a hash tag sign and to end with a node labelled with a dollar sign. When opening a GML file, these extra nodes are therefore created first, and the main row as specified in the GML file is added between them as one possible way of traversing the graph from the hash tag node to the dollar sign node. Then the info blocks are iterated over, and their contents are added as edges and nodes.

3.4.1. Randomly Generated Graphs

The Graph Merging Library also contains a function for the automatic generation of random graphs, which can be used to quickly generate input for testing out the provided functions.

3.5. Core Assumptions of GML

There are several core assumptions which graphs need to fulfill to be used within the Graph Merging Library. These are chosen to simplify working on the graph as opposed to working on more general arbitrary graphs, while allowing for enough freedom to encode actual real world data.

Graphs used in the Graph Merging Library are generally considered to consist of nodes each of which contains a label and directed edges.
Every node of the graph needs to be connected to every other node of the graph, but not every node needs to be reachable from every other node. This means that every set of two nodes needs to be connected by a series of edges, but not all of them need to face in the same direction. Also, no two sets of directed edges from A to B and from C to D are allowed to exist within the same graph for which both $A = C$ and $B = D$ are true.
For a graph to be considered within the Graph Merging Library, it is not allowed to contain infinite loops. These are any structures in which there exists a node which can be reached by starting at it and following a series of edges along their stated direction.

Graphs are also assumed to have exactly one node with the label "$" which is the start of

each path traversing the graph. This means that there is only one node with no incoming edges, this node has the label "$", and it is the only node in the graph with this label. Furthermore, each graph is assumed to have exactly one node with the label "#" in which each path traversing the graph ends. This can be reformulated as there being only one node without any outgoing edges, this node having the label "#", and no other node in the graph having this label.

If graphs are supposed to be merged together, then another core assumption is added, which states that no node splitting is allowed to be needed between these graphs. That is, at no point within the merging process shall it be necessary to connect the two graphs through more than one edge.

3.6. Visualizing the Graph

After opening the graph, it can be helpful to show it to the user for quick and easy visual inspection. We therefore included a small visualization function within the Graph Merging Library, which is capable of displaying simple graphs nicely. Algorithm 3.1 shows how this function works in general. Having a closer look at line 1 of the algorithm, it can be seen that it constructs the main row by starting in the first node, which is assumed to be labelled with the hash tag symbol. In line 6 of the algorithm it can then be seen that to construct the main row starting with this node, the algorithm always jumps into the first of the successors of the current node.
Therefore, before a graph can be passed to the visualization function, it needs to be ensured that its hash tag node is stored in the very first position and that the node labelled with the dollar sign can be reached by always travelling along the first of the successors. A separate function called `makeAutomatonPretty` is used to ensure that this is always the case, by traversing all possible paths and picking one path which leads to the dollar sign node. In the options it can be chosen whether this function should find one of the shortest paths, which is faster, or one of the longest paths, which leads to a better visualization. When it is instructed to find one of the longest paths, it will however still ignore paths including loops, as it might otherwise be trapped in the loop indefinitely, trying to find yet longer paths. An example for how this choice affects the visualized graph can be seen in figure 3.5. The graph nodes are then adjusted in such a way that following the first successor of each node upon starting at the hash tag node leads directly to the dollar sign node.
It is also notable how exactly the iteration over the nodes works out. In line 10 we iterate over all nodes that have been added to more_nodes, an array which keeps track of nodes which we have not fully worked on yet. This means that we may have already drawn the node, but we have not yet drawn all the incoming edges, and we may possibly not even have drawn the node itself. The loop in line 15 then iterates over all the first successors of that node, and adds them to an array called path, until a node is encountered that has already been drawn. Finally, in line 22, the entire path is drawn together.
Algorithmically it would be much more straightforward to simply iterate over all nodes

Algorithm 3.1 Visualizes a graph by first displaying one path from the starting node to the end node, referred to as main row, and then adding alternative paths around the established core.

```
1:  current_node ← graph[0]                                  ▷ Display main row
2:  draw current_node
3:  more_nodes ← [ ]
4:  while current_node.label is not $ do
5:      append current_node to more_nodes
6:      current_node ← current_node.successors[0]
7:      draw current_node
8:  end while
9:
10: for all more_nodes as path $ do                          ▷ Display other paths
11:     current_node ← path
12:     if current_node has not been drawn then
13:         node ← current_node
14:         path ← [ ]
15:         while node has not been drawn do
16:             append node to path
17:             node ← node.successors[0]
18:             if node has not been in more_nodes before then
19:                 append node to more_nodes
20:             end if
21:         end while
22:         draw nodes in path together
23:     end if
24:     for all current_node.predecessors as predecessor do
25:         draw edge from predecessor to current_node
26:     end for
27: end for
```

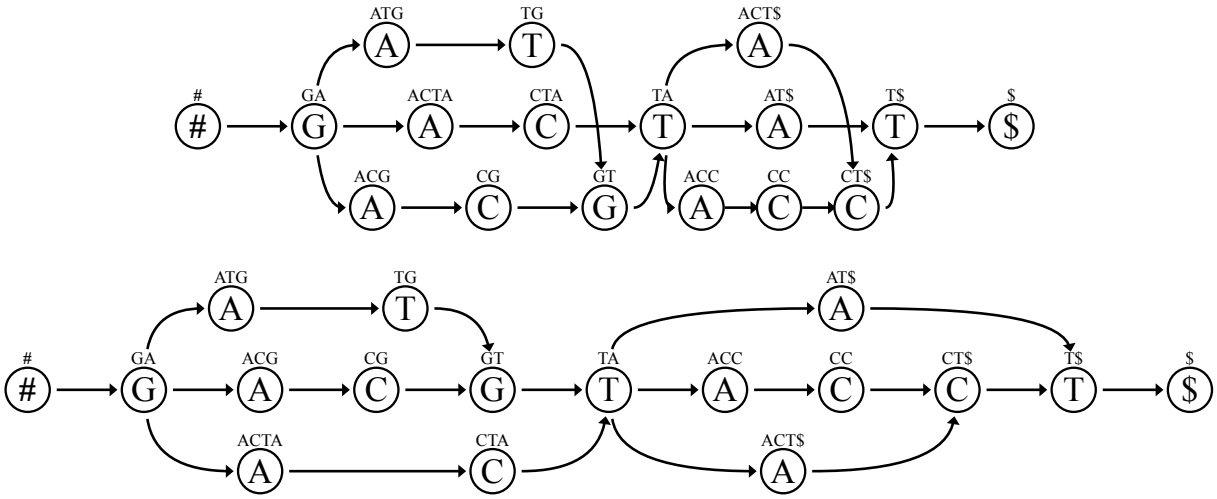

Figure 3.5: Two visualizations of the same graph. The visualization on the top uses the shortest possible path as main row, while the visualization on the bottom uses the longest possible path.

and draw them when we encounter them, but in that case we would not have a good idea about where it might make sense to draw them. The approach of iterating over nodes that have not been worked on and nesting into this an iteration of the successors of that node allows us to instead draw entire parts of paths at once, so that we have a good idea where exactly to put the nodes on the display when drawing them, as they can simply be arranged next to each other.

3.7. Ensuring the Graph is Reverse Deterministic

Having displayed the graph, we now continue working on it to ultimately be able to work with it in the extended XBW format.

For any node within the graph, we can construct a prefix, which is the sequence of its own label concatenated with the labels of the nodes which are traversed when exiting that node. The extended XBW compression scheme for graphs as proposed by Sir relies on the possibility of ordering all nodes of a graph alphabetically, based on their prefixes. After opening a file format that can contain arbitrary graphs, it is therefore first of all necessary to ensure the graph is reverse deterministic, as otherwise at least two nodes could not be unambiguously sorted against each other. File formats which can contain graphs which are not reverse deterministic include FASTG, GFA, GML and the bubble format, while the FFX format guarantees that the stored data is reverse deterministic as its entire encoding paradigm otherwise would not work.
To understand why a graph not being reverse deterministic would lead to at least two nodes not being able to be unambiguously sorted by their prefixes, we can recall the definition of a reverse deterministic graph:

Definition A graph is reverse deterministic if and only if each node has no two predecessors with the same label.

From the definition it can be seen that a graph not being reverse deterministic means that there is a node A which has at least two predecessors B and C which share a label. As B and C share a label, their prefixes both start with the same label.
Without loss of generality we can now focus on one of the two nodes B and C. As B is a predecessor of A, we can see that the prefixes of B either continues with the prefix of A, or that it will not be able to continue after the first character as there is another successor of B which has a different label than A does.
The same holds true for C, such that both B and C have labels starting with the same character and are of length 1 unless they are that character concatenated with the prefix of A.
Now, in case of one of the prefixes just having length 1, we cannot sort B and C unambiguously, as their prefixes are the same for all given characters. On the other hand, if both prefixes have length above 1, then they are both the same character concatenated with the prefix of A, as we can again not sort B and C unambiguously. Therefore, a reverse deterministic graph invalidates the assumption that we can sort all of its nodes unambiguously alphabetically by its prefixes. □

The program therefore first needs to check if the opened graph is reverse deterministic. This can easily done by comparing the labels of every nodes' predecessors, as can be seen in algorithm 3.2.

Algorithm 3.2 Checks if a graph is reverse deterministic.

```
1:  for all graph.nodes as node do
2:      encountered_characters ← []
3:      for all node.predecessors as predecessor do
4:          if encountered_characters contains predecessor.label then
5:              return false
6:          else
7:              append predecessor.label to encountered_characters
8:          end if
9:      end for
10: end for
11: return true
```

If the graph is found not to be reverse deterministic, then it needs to be adjusted until it becomes reverse deterministic. It is hereby important that any change to the graph does not change the language that the graph realises, which represents all genomic strings that are encoded within the graph.
To change the graph in this way, three different operations can be used on the graph. These three operations are to merge two nodes, to move an edge from one pair of nodes to another pair, and to split a node. Of course, not all nodes can be merged with each other and not all edges can be moved around without changing the language that the graph represents, which is why a special algorithm needs to determine which operation

to choose next and how to apply it.

3.8. Prefix Sorting

The next step is to ensure that all the nodes can actually be sorted by their prefixes. If this requirement is fulfilled, then we call the graph prefix sorted.
The graph is already reverse deterministic, but that does not mean that all the nodes necessarily have unique prefixes which make them unambiguously sortable. An example for a graph which is reverse deterministic but contains several nodes with the same prefixes can be seen in figure 3.6. If none of the nodes in the graph had more than one outgoing

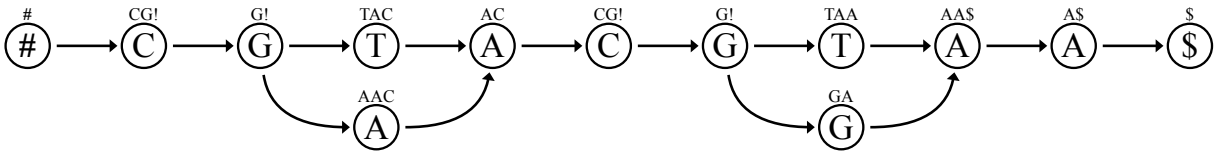

Figure 3.6: Reverse deterministic graph that is not prefix sorted. Among others, the two nodes with label "C" share the same prefix.

edge, then the graph would already be prefix sorted, as every prefix would end with the dollar sign and have this dollar sign in a different position than any of the other prefixes would. However, as the nodes in the graph we consider can actually have several outgoing edges, it is possible that they lead to nodes which have different labels. In this case, we write a special character such as the exclamation point at the character in the prefix, to note that this character can not be unambiguously determined.
The process of achieving a prefix sorted graph is therefore based on iterating through the nodes and reducing the amount of prefixes which end on exclamation points, as can be seen in algorithm 3.3. In line 18 of this algorithm, a node with more than one successor is split into several nodes. Such a node needs to exist there, as we reach this point in the algorithm due to a prefix not being unambiguously constructable, and the only way for this to happen is if a node has several successors with different prefixes. Therefore, we must be able to find a node with several successors for which splitting it enables us to build longer prefixes.
The exact approach for splitting a node N is to replace N by as many new nodes as N has outgoing edges. Every one of the nodes replacing N has exactly one outgoing edge, leading to one of the successors of N. In this fashion, every successor of N obtains one of the nodes replacing N as predecessors. Every one of the new nodes also obtains incoming edges to all the nodes that are predecessors of N. Finally, every one of the nodes replacing N is given the label of N. The process of splitting a node is shown in figure 3.7.

It should be noted that we do not wish to remove all exclamation points entirely, as it is enough for a prefix to be so long as to be unambiguously sortable against all other prefixes of nodes in the graph. If a prefix is long enough to ensure this, then it is completely irrelevant whether it eventually ends in an exclamation point or in a dollar sign.

Algorithm 3.3 Prefix sorting a graph by splitting nodes with prefixes that are not unambiguously sortable.

```
 1: graph_is_not_prefix_sorted ← true
 2: while graph_is_not_prefix_sorted do
 3:     graph_is_not_prefix_sorted ← false
 4:     for all this_node in graph do
 5:         this_node.prefix ← this_node.label
 6:     end for
 7:     for all this_node in graph do                    ▷ Check for all nodes...
 8:         same_as ← []
 9:         for all that_node in graph do          ▷ ... if any node has the same prefix
10:             if this_node.prefix = that_node.prefix then
11:                 append [that_node] to same_as
12:             end if
13:         end for
14:         if length of same_as > 1 then
15:             for all node in same_as do      ▷ Iterate over nodes with the same prefix
16:                 first_node_label ← next character to append to node.prefix
17:                 if first_node_label cannot be found unambiguously then
18:                     split the last node whose label was added to the prefix and which
19:                         has more than one successor
20:                     graph_is_not_prefix_sorted ← true
21:                 else
22:                     append first_node_label to node.prefix
23:                 end if
24:             end for
25:         end if
26:     end for
27: end while
```

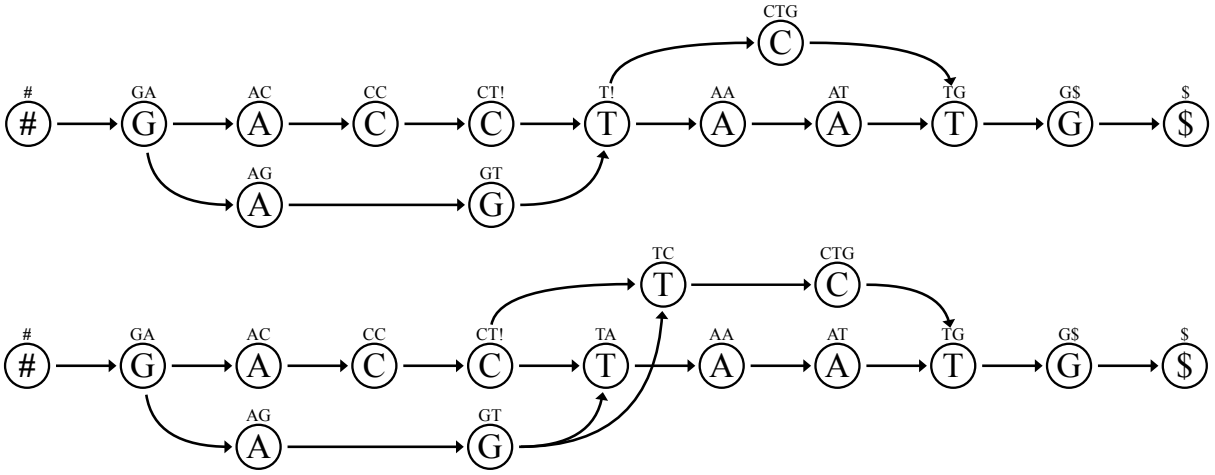

Figure 3.7: Node splitting example. On the top, the graph is visualized before the splitting of the central node with label "T" and prefix "T!". On the bottom, the same graph can be seen after node splitting occurred. The node has been replaced with one node with prefix "TC" and one node with prefix "TA". Both nodes have all the incoming edges that the original node had.

3.9. Creating a Node Table

Instead of directly generating a flat XBW table from the prefix sorted graph, We decided to first generate a node table. This is a table that shares important characteristics with a flat XBW table, but which is easier to understand and to use as it is closely related to the graph. Using the visualized graph to understand the corresponding node table can therefore be very helpful.

Definition An XBW node table is a table with rows for the BWT, the prefixes and the values of M and F, representing a finite graph with each column of the table corresponding to exactly one node of the graph.

The columns in a node table are sorted alphabetically by the given prefix. For any column C representing node N in the graph, the value of the BWT in C represents the labels of the nodes preceding N. If several nodes with different labels are preceding the node N, then the BWT value takes on the concatenation of all these values. To make it clearer that a BWT containing several characters actually stands for any of these characters, instead of standing for the string formed by the characters, we decided to display BWT values containing more than one character separated by pipe characters, as can be seen in figure 3.8 which is represented by node table 3.1.

The value of the prefix of C is precisely the prefix of node N in the graph, which is a string starting with the label of N and then containing the labels of the following nodes in sequence of encountering them traversing the graph outwards from N. The value of M is a string whose length corresponds to the amount of successor nodes of N, or equally the amount of outgoing edges of node N. This string always starts with the character

Table 3.1: Node table corresponding to the graph in figure 3.8.

G	G	G	G	A	C|G	G|T	#	G|T	T	T	T	T	C	C	C	$	**BWT**
$	A	CG	CT	G$	GA	GCG	GCT	GGA	GGC	GGG	TGC	TGGA	TGGC	TGGG	TT	#	**Prefix**
1	1	1	100	1	1	1	1	1	1	1	1	1	1	1	10	1	***M***
1	1	1	1	1	10	10	1	10	1	1	1	1	1	1	1	1	***F***

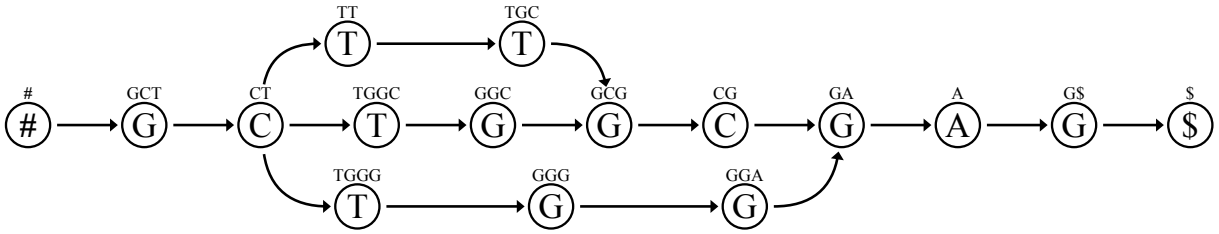

Figure 3.8: Graph corresponding to node table 3.1.

"1" and is then followed by as many "0" characters as are necessary to achieve the desired length. Similarly, the value of F in column C is a string whose length corresponds to the amount of predecessor nodes of node N.

3.10. Merging Node Tables

The main aim of the Graph Merging Library is to provide algorithms to merge and fuse flat XBW tables. However, working with flat XBW tables directly is rather cumbersome, and it can therefore help to first investigate the simpler case of merging two node tables.

3.11. Creating a Flat Table

Having understood how merging several node tables works, we can now carry on with the actual flat XBW tables. Encoding the same graphs both as flat XBW tables and as node tables shows that flat XBW tables are much smaller and therefore easier to store. The biggest differences between the two kinds of tables, which can both be used to encode genomic graphs, are that flat tables do not contain the potentially enormously big prefix cells of node tables, and that flat tables do not require any overhead to keep track of column boundaries within a row. While node tables can contain strings of various sizes in all their cells, flat tables contain exactly one character in each cell, such that an entire table row can be stored as string without losing information about the cell boundaries.
This smaller size comes at the cost of simplicity. Within a node table, each column clearly corresponds to a node within the graph, and a simple manual inspection of the table can lead to various insights into the graph. In a flat table however, not every column describes and entire node within the graph, as some columns only describe parts of nodes,

as e.g. a node with several predecessors can have information about its BWT spread across several columns in the flat table. In addition to that, we should not even think about whole columns in the flat table necessarily, as we need the M and F rows to find the corresponding cells across different rows, which are not necessarily located in the same columns.

To convert an XBW node table into a flat table, we need to convert the data which is now in the format of a node table into a flat one, a process which can be seen in tables 3.2 and 3.3. For the BWT, M and F rows, the process is exactly the same. We join

Table 3.2: XBW node table before conversion to flat table. Table 3.3 shows the corresponding flat table.

G	T	G	G	G	A	T	T	#	A	A|G	A|G	A|C	$	**BWT**
$	AA	AG	ATA	ATC	ATG	C	G$	GA	GT	TA	TC	TG	#	**Prefix**
1	1	1	1	1	1	1	1	100	10	1	1	1	1	***M***
1	1	1	1	1	1	1	1	1	1	10	10	10	1	***F***

Table 3.3: Flat XBW table after conversion from node table. Table 3.2 shows the corresponding node table.

G	T	G	G	G	A	T	T	#	A	A	G	A	G	A	C	$	**BWT**
$	A	A	A	A	A	C	G	G	G	G	G	G	T	T	T	#	**FC**
1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	1	***M***
1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	***F***

all the cells together to achieve one string containing the entire row, and then split this row in between every character. If there were pipe characters in between some of the characters, such as for visualization purposes within the BWT cells, then we omit them in this process.
The process for the prefixes is a little bit different, as the flat XBW table does not contain the whole prefixes due to their immense size. The flat XBW table can however be thought of as containing FC, the first column data (where the first column refers to the first column of the alphabetically sorted cyclic rotations, not to the first column of this table.) As the FC field corresponds to the label of the node, and as the prefixes in the node table begin with the labels of their respective nodes, the two rows are intimately linked and the FC row can be constructed from the prefixes in the node table. For the construction of the FC row, it is also necessary to consider the amount of outgoing edges of a node, which is encoded in M. In total, to generate the FC row in the flat table based on the prefix and M rows in the node table, we start with an empty string for FC and iterate through the prefixes in the node table. For each prefix, we read out the first character of that prefix. We then add this first character as often to FC, as the contents of the M cell in the corresponding column are long. So if the value of M is 100, then the first character of that prefix needs to be added three times to FC.
Having constructed the BWT, FC, M and F rows from the original node table, we have

now created a flat XBW table which represents the same graph that was represented by the original table.

3.12. Working on a Flat Table

Before considering the rather challenging problem of merging several flat XBW tables, we should first investigate how work within such flat tables can be performed in general. In particular, it is helpful to understand how the functions LF and Ψ can be used to find the predecessors and successors of nodes, respectively.

3.13. Merging Flat Tables

The merging of flat XBW tables is in principle similar to the merging of node tables. We again iterate over all nodes of all involved graphs and construct a new table based on them. However, a big difference to the merging of the node tables is that we no longer have a data structure available to store the computed prefixes of all columns. This is intentional, as storing all of these prefixes in memory would be prohibitive in a real-world scenario. The implication of not having the prefixes readily available are that we can no longer keep track of problematic nodes that easily. We also cannot merge the separate tables together as they are and work on the node splitting to expand the prefixes after the merge, as the flat XBW format relies heavily on the ordering on the columns. We simply cannot guarantee that we will order the columns correctly if we do not have expanded prefixes in the first place.
We therefore decided to implement the merging of flat XBW tables in two steps, with the first being an iteration over all columns of the tables that are supposed to be merged, with the aim of determining which prefixes will need to be expanded. The corresponding nodes are split and the prefixes recalculated until no further node splitting will be necessary when the actual merge occurs.
The second step is then to carry out the merging. This in itself is complicated by the fact that in flat XBW tables, entire columns do not necessarily form a union. That is, in the node table, each column consisting of an entry in the BWT, a prefix, and values for M and F, represents one node in the graph, and can not be taken apart. However, in the flat XBW table, column do not form such a union. Instead, an F value of 0 shows that the BWT of this column forms a union with the BWT of the column to the left, and an M value of 0 shows that the first column value of this column forms a union with the first column value of the column to the left.

3.14. Fusing Instead of Merging

As shown in the previous section, merging flat XBW tables is rather complex. It requires a lot of computation and due to a lot of node splitting necessary to ensure unambiguously sortable prefixes along the entire merged graph can result in large amounts of node splitting. This leads to an increase in file size as opposed to the total file size of the individual graphs that are merged.
To explore how these problems might be avoided, We decided to implement another way to combining several flat XBW tables. In this way, consecutive graphs encoded as flat XBW tables are fused together on their ends, meaning that the dollar sign node of one graph is connected with the hash tag node of the next. This connection is not directly achieved within the data structure itself, but it is instead performed by the program working on the data structure, which in itself just contains several completely regular flat XBW tables with one hash tag and one dollar sign node each.

Implementing the fusing behaviour is not particularly difficult, as the only new occurrence to look out for is a possible spillover in which we leave one graph and move on to the next. In that event, the program working on the fused XBW data structure needs to automatically jump over the hash tag and dollar sign nodes, as internal hash tag and dollar sign nodes should not be accessible to the outside.

4. Results

In this thesis, existing graph reference approaches have been compared and new algorithms have been implemented to help the community effort of starting to use reference graphs more widely.

4.1. Formats for Genomic Graphs

We tried out different formats in the course of this thesis, and noticed that indeed some do seem more appropriate for representing genomic graphs than others.
Surprisingly, implementing an algorithm to open a file in the seemingly simple bubble format was actually not easier than implementing more advanced formats like FASTG, GFA and GML.

We also noticed that none of the considered formats currently offer inbuilt compression techniques. Changing this however would not be very difficult. Especially in the FFX format, which mainly contains a BWT and two bit-vectors for each data block, a rather simple compression method would be to save both the BWT and the bit-vectors using run-length encoding. Such an approach would still keep the format simple enough to be implemented in various tools in the future, while offering a significant file size reduction. This reduction is especially strong when data is encoded that contains long linear sections interspersed with some short graphs, as both of bit-vectors will then contain very long runs.
It is of course also possible to create even more compressed versions of the different formats by not saving the contents of the files as ASCII text, but instead as bitwise data structures that can e.g. represent each of the nucleotides A, C, G and T with just two bits.

4.2. Merging XBWs

Using the Graph Merging Library, we have explored several ways to merge XBWs. As could be expected, merging two node tables is a much simpler process than merging two flat tables, as a node table is much less compressed, and working on it is simpler in general. The node table not being as compressed as a flat table though means that it would not be practical in a real-life scenario to merge these kinds of tables.

However, even when choosing to merge two flat XBW tables, the merge result can become unnecessarily large.

4.3. Fusing XBWs Instead of Merging Them

When merging flat XBW tables together, it is common that node splitting will result in a merged graph with a size larger than the total size of the original graphs. Fusing flat XBW tables together omits the need for this increased file size, which not only makes it easier to store the data, but also means that working on it is quicker as less data means that fewer operations are necessary to construct distinct prefixes, to traverse all possible paths, and so one.

However, fusing flat XBW tables also has downsides. One of them is related to indexing. When working with a merged graph, every column in the graph has a clear integer index, so that locations within the entire graph can simply be referred to with a single integer value. With a fused XBW however, in addition to specifying a column within the table, it also needs to specified which table is addressed, necessitating pairs of integers. If particularly large list of locations needs to be created, this difference of two integers instead of one could become problematic.
Also, it is algorithmically more complex to work with fused graphs, as every algorithm working on the graph needs to be prepared to handle spillover scenarios, in which the end of one of the fused graphs is reached and the execution needs to continue within the next graph.
These spillover scenarios might also severely limit the otherwise seemingly great possibilities for splitting the computation of fused graphs across different cores in a distributed system. If each processor core is supposed to work on exactly one graph, but spillovers are happening frequently, then some cores would stop having work to do as no further work would be going on within their data block, while others would have to do twice as much work by caring for their own work before working on the activity that spilled over from the other data block.
Finally, setting up such a system in the first place might be difficult to do, as a fused file would most likely contain large areas of linear data, fused together with short blocks of graph data. Therefore, assigning one data block to one core and letting it perform work in this straightforward fashion would lead some cores to have to do much more work than the others, ultimately leading to many idle cores waiting for a few to finish which were given much more work to do in the first place. This is obviously not desirable in a multi-core system, in which ideally all the processor cores should be working for the same time and then be done together in the quickest time possible.

However, at least some of these problems could be overcome fairly easily. Spillover scenarios as an example, which with a naive implementation would possibly lead to some cores having no work to do after a spillover happened while other cores would have to do twice as much, could be avoided by giving the cores access not only to their own data

blocks, but also to the data in the surrounding blocks. Then each core could handle and complete its own spillovers, without requesting assistance from its neighbours.

4.4. Automated Tests

In general, we can confirm how well the Graph Merging Library functions by checking that for a certain set of predefined inputs, the correct outputs are generated. For that purpose, we provided several kinds of automated tests for both the graph and XBW generation from an input, as well as the merging of several input graphs in different ways.

4.4.1. Predefined Tests

The basic approach for testing the validity of the GML algorithms is to run a set of predefined tests, for which the correct solutions are known. Doing so will show a list of tests and the test results, which can be one of four states. Figure 4.1 shows how these predefined tests can be executed in the graphical user interface of the Graph Merging Library.

The ideal result is a *success*, in which the algorithmically generated result agrees with

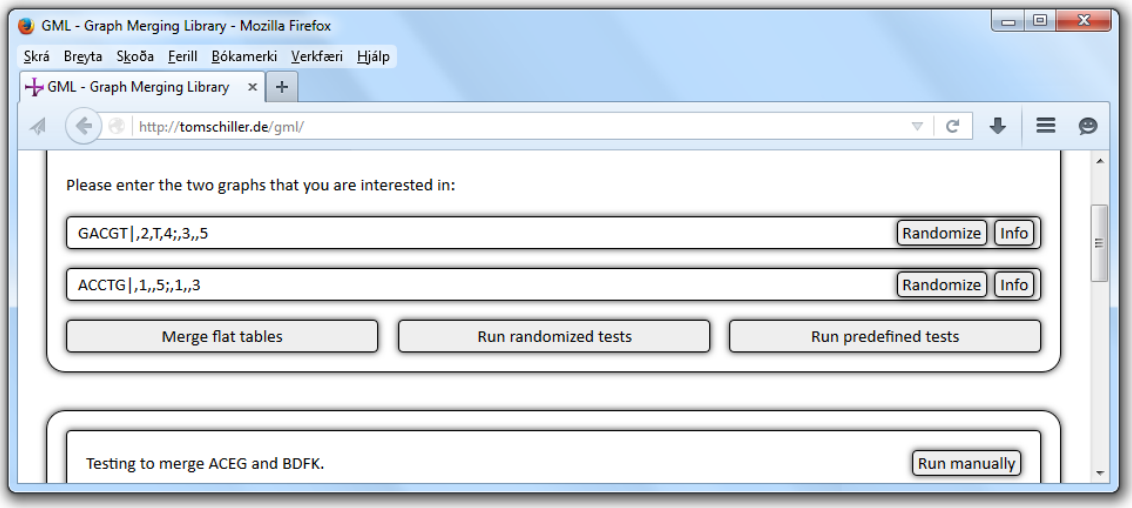

Figure 4.1: GML screenshot with row of three main action buttons. Clicking on the rightmost button starts the predefined tests, while clicking on the button on its left starts the randomized tests.

the expected result of the test.
Instead, a *failure* can be reported if the result as generated by the GML algorithms differs from the expected result.
Finally, the label *crash* is associated with a test which causes the entire program to crash

completely, without generating any useful results whatsoever. Ideally, no input data should ever be able to cause this to happen, but if it should happen, then a crash would be reported to alarm the user to it.
The representations of all these result types in the GML interface can be seen in figure 4.2.

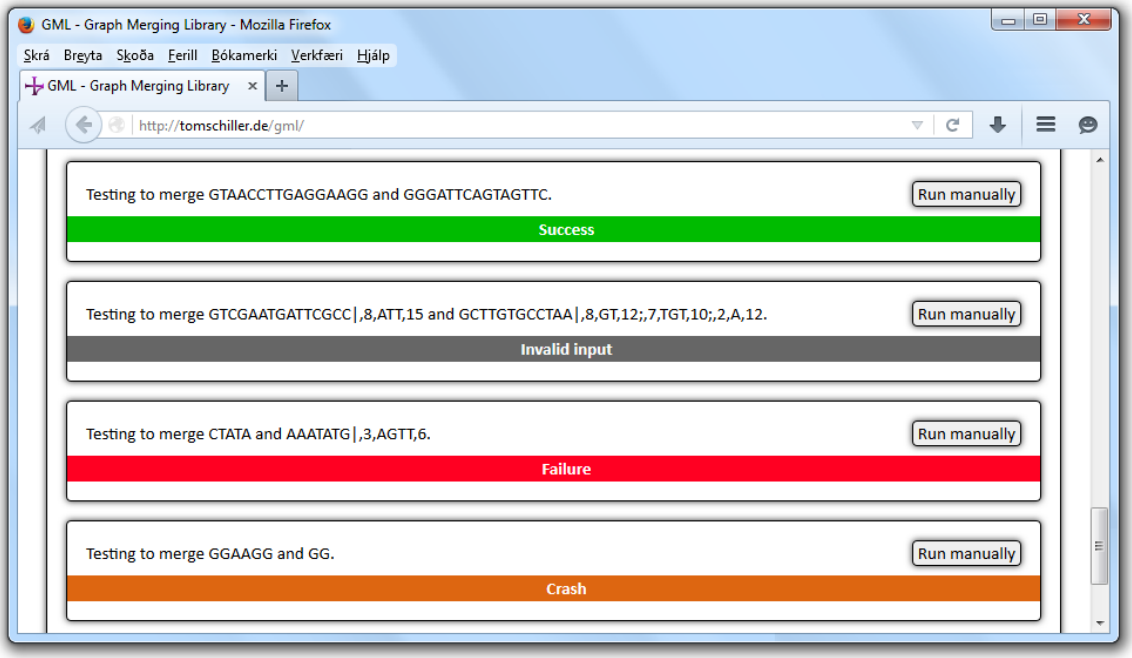

Figure 4.2: Different test results in the GUI of the Graph Merging Library.

4.4.2. Randomized Tests

The predefined tests are intended as a method for gouging how well the algorithms are functioning, but it could be argued that they might be hand-picked to only show positive outcomes. Therefore, a second mode of testing has been implemented in which random test data is created during the execution time of the program. However, generating such random test data automatically means that it is difficult to ensure that the input conforms to the core assumptions as formulated in section 3.5. It is in particular difficult to randomly generate test data for which no spillover in the splitting of nodes across several graphs which are supposed to be merged could possibly happen during the merging process. Opposed to that, it is straightforward to ensure that the input graphs contain no loops.
[1] It is therefore necessary for the program to ensure that the input can actually be worked

[1] This can be achieved by starting out with a random sequence of characters as one path from the start to the end of the graph. Then new paths can be added one at a time to the emerging graph, with each new path from node A to node B only being allowed to be added if B could already be reached from A before the new path was added.

on, and to give out a warning if not. That is why another possible test result with the label *invalid input* can be reported. This occurs when the program determines that the input does not conform to the previously formulated core assumptions.

4.4.3. General Note on Testability

The idea behind the automated tests and especially behind the randomized tests is to let the Graph Merging Library test itself, so that it can easily be seem whether the algorithms provided within it function well or badly.
However, considering a merge test with several randomized inputs shows a mismatch between the expected result and the actual result, it can be reported as a failure because the actual result of the algorithm failed to be the expected result. On the other hand, the same behaviour can also occur if the randomized input does not conform to the basic assumptions.
In general, it is not always entirely clear whether the program failed to provide the correct answer because the input was invalid, or because a bug prevented the generation of the correct output. The reason for this is that determining whether the input is invalid or not is part of that very same source code, and if a bug prevents it from correctly working, then the this may be determined wrongly. In the opposite way, a bug might also prevent an actually invalid input from being flagged that way, resulting in a reported error which actually is no error at all.
Overall, this means that tests which are reported as failures should be manually investigated more clearly to better understand the underlying cause of the categorization, and to make sure that it is not simply a case of the input not honouring the core assumptions.

4.4.4. Test Results

Running the over 100 predefined tests for the XBW creation, node table merging and flat table merging results in successes for each of them, with no failures or crashes being reported. As the tests have been picked to include various challenging situations in which complicated node splitting behaviours arise and in which special sorting mechanisms need to be used to ensure the correct outcomes, it can be seen that the algorithms appear to be working fine.
It should of course be noted here that no amount of tests can ever truly prove these algorithms to truly be working correctly, as any amount of tests performed is necessarily finite, while there are infinitely many inputs that could theoretically be specified, such that it will never be possible to check every single input that there is.

5. Conclusions

Aligning reads from human DNA to graph references rather than to string reference sequences still offers many interesting problems. However, the first steps on the way to using reference graphs in real-world applications have been made.

5.1. Discussion

Despite a lot of work already having been done to explore ways of using genomic graphs within the field of human DNA analysis, using reference graphs rather than reference strings still is a difficult proposition and only few tools exist that incorporate such methods. It could therefore be argued that the known advantages of using reference graphs do not outweigh the tremendous difficulties that have to be overcome in order to be able to use them on a large scale.
After all, the reason why reference graphs are proposed to be used rather than reference strings is to improve alignment results and to simplify the calling of known variants. But at least in the beginning when only small local graphs are incorporated into largely linear references, the improvements are likely to be very slim. Likewise, it could be questioned whether simplifying the calling of variants really justifies complicating matters so much in general by the introduction of these new graph structures.

However, we would argue that in the long run, reference graphs will come to be used. With every new individual DNA that is constructed, more insight is gained into variations which could be quite common among populations. Discarding all of that data in favour of a simpler but ultimately flawed approach will not work forever, as over time better alignment results will come to be expected.
Therefore, it seems inevitable to work on reference graphs eventually, and it seems only logical to start that work as soon as possible, rather than investing more time and resources into improving current alignment tools which are likely to be drastically modified to incorporate reference graphs in the future anyway.

5.2. Future Work

There still remains a lot of future work to be undertaken to actually use the already existing and the here newly proposed algorithms in real-world tools of the analysis pipeline. As various different tools are in use for specific problems, no one implementation is likely to achieve this. Instead, the community at large needs to start incorporating the ideas of using graph data rather than sequential data into more and more of the existing tools.

6. References

a.

b.

a.

b.

c.

d.

e.

A. Appendix

A.1. Resources and Licensing

Publicly available genomic datasets as well as automatically generated data were used to implement the new graph reference approaches and to analyse the results of the implementations.

Recently, there has been a push towards implementing more open source scripts and programs within the scope of bioinformatics (MAN).
All of the software developed while working on this thesis is open source and distributed under the terms of the Creative Commons Attribution Non-Commercial License (`http://creativecommons.org/licenses/by-nc/3.0/`), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the original work is properly cited. For commercial re-use, please contact `moyaccercchi@hotmail.de`.
The source codes together with the newest version of the thesis itself can be accessed at `http://tomschiller.de/graphalign`.