Reference Graph Construction and Merging for Human Genomic Sequences

Tom Willy Schiller

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

REFERENCE GRAPH CONSTRUCTION AND MERGING FOR HUMAN GENOMIC SEQUENCES

Tom Willy Schiller

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

Advisors
Páll Melsted
Kristján Jónasson

Faculty Representative
Birte Kehr

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 Genomic Sequences
Reference Graphs for Human Genomic Sequences
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 Genomic Sequences, 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 instead using a graph reference are explored.
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 for working with genomic graphs 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

absolute indexing: Method of indexing in flat XBW tables. Its value does not depend on a choice between the FC or BWT row. See page 19.

bubble: Simple encoding of short alternative paths in a genomic graph, such as A[C,G]T, which refers to both ACT and AGT.

BWT: The *Burrows–Wheeler Transform* is a string compression technique (Burrows and Wheeler, 1994) which is used during the read alignment. For a good introduction to the usage of the BWT in bioinformatics, see Compeau and Pevzner (2014, chapter 7).

BWT-indexing: Method of indexing in flat XBW tables. Its value is based on an index in the BWT row. See page 19.

end node: If a graph has a node with label "$" having no outgoing edges, then we call this the *end node* v_e of the graph. See page 32.

FC-indexing: Method of indexing in flat XBW tables. Its value is based on an index in the FC row. See page 19.

flat table: A *flat XBW table* is a data structure consisting of a BWT string, two bit vectors and some additional information. This structure completely represents a graph and can be used to store the graph in a file or to perform work on it, such as navigating between nodes and finding texts within the graph. See page 16.

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.

node table: An *XBW node table* is a data structure consisting of a BWT row, a prefix row and one or two bit vector rows. Hereby the cells within each column do not need to have the same lengths. This structure completely represents a graph and can be used to store the graph in a file or to perform work on it, such as navigating between nodes and finding texts within the graph. See page 15.

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.

rank: The *rank* operation counts how often a given character occurs in a sequence before a certain index. See page 18.

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.

select: The *select* operation returns the index in a sequence at which a given character occurred a certain amount of times. See page 18.

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.

start node: If a graph has a node with label "#" having no incoming edges, then we call this the *start node* v_s of the graph. See page 31.

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, Rosemary Milton and Karl Helgason 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 develop algorithms for merging and fusing flat XBW tables, which will be introduced in section 2.7.2.
We developed these algorithms 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 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.

1.1. The Role of the Reference

Many technological advances in recent years have already made it possible to sequence the whole genome of an individual for under $ 1000 USD. The process currently most often used to do so is based on a machine sequencing several strands of the individual's DNA, short for deoxyribonucleic acid. The sequencing process however does not directly produce a data structure representing the entire individual's DNA. Instead, it produces a large collection of short pieces, called reads. Such reads are strings consisting of the four letters A, C, G and T, which represent the four nucleobases adenine, cytosine, guanine and thymine, respectively.
Having sequenced the reads, a pipeline of several software tools is used to align them to a reference genome. The aim of this alignment is to either generate the full individual DNA or a list of variations from the reference that has been used.
However, the sequencing process still generates many errors. While work is continuously being undertaken to improve the quality of the sequencing results, it is not expected that completely error-free reads are going to be sequenced from an individual's DNA soon. Therefore, the software tools used for working on the reads also need to continuously be improved, to make better use of the data generated by the sequencing process. One way of doing so is by aligning the sequenced reads against a reference graph rather than a reference string against.
Such a reference string 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 genomic 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

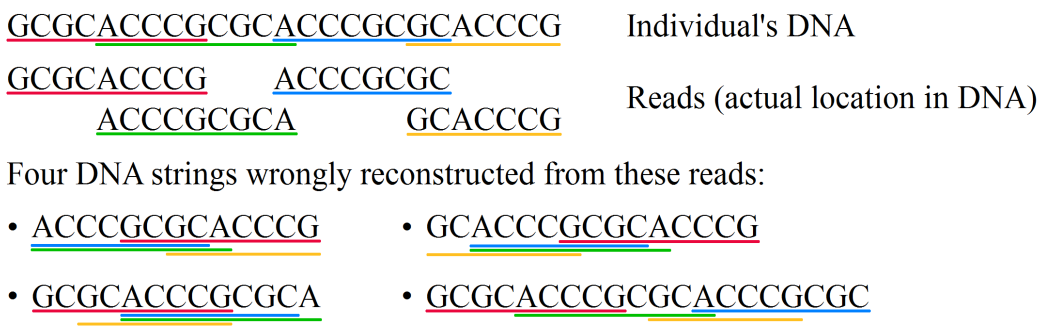

Figure 1.1: Problems with assembling an individual's DNA from small reads without a reference. Four possible reconstructions of the individual's DNA are shown, which all correspond to the observed reads, but are different from the actual DNA of the individual.

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 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

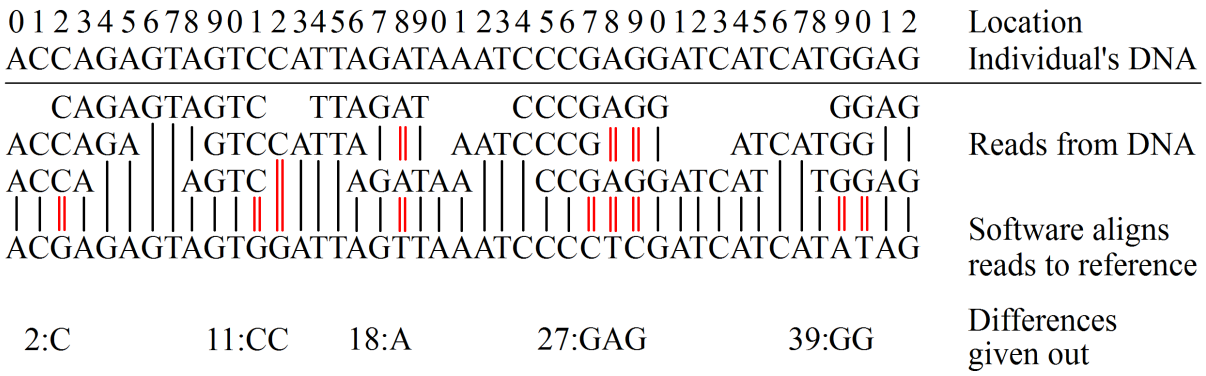

Figure 1.2: General process for generating a list of variations between the individual DNA and the reference.

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.

[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/`

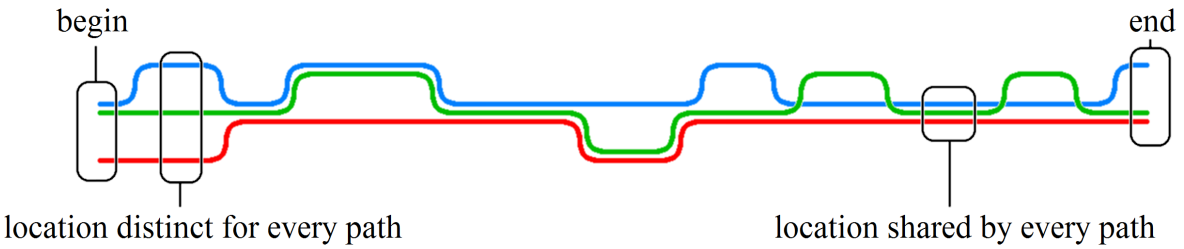

Figure 1.3: Schematic visualization of three paths through a variation graph, each representing an individual's genome.

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 (Albers et al., 2011).

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 (Cooper, 2015).
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. We will first focus on the Burrows–Wheeler Transform, or BWT, for strings. It is the basis for the more advanced approaches for genomic graphs considered later on, which use the extended Burrows–Wheeler Transform, or XBW.

2.1. Burrows–Wheeler Transform for Strings

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.
We therefore wish to pre-process either the reads or the reference in a certain way, ensuring that the work which needs to be performed in each step gets reduced. Practically, it often makes sense to choose to pre-process the reference, as the same reference can be used to align reads for several individuals, and therefore the computationally expensive pre-processing only needs to be done once.

A common way for pre-processing the reference is to apply the BWT to it, and to then use it in conjunction with a suffix array. This allows us to perform fast lookups of patterns in the reference, which is exactly what we want to do to find the locations at which we can align the reads.
The BWT, or Burrows–Wheeler Transform (Burrows and Wheeler, 1994), first emerged as a tool used in string compression techniques. It is a reversible reordering of 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 or other encoding schemes. A simple example of this behaviour of achieving more easily compressable strings through the application of the BWT can be seen in table 2.1. As the transformed text contains all the information necessary to recreate the original text, it is not necessary to store the original text at all.

A simple algorithm to generate the BWT of a string uses the concept of *cyclic rotations*. The nth cyclic rotation of a string is a reordering of the string in which its first n characters are taken away from the beginning of the string and are instead inserted at its end.
To generate the BWT of a string $s = c_1c_2...c_k$ of characters c_i, we can generate all the cyclic rotations from $n = 0$ up to $n = k - 1$, and put the resulting strings on a list.

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**

We then sort this list alphabetically, and write out the resulting list of strings in such a fashion as to have each of the strings on its own row, exactly underneath each other. This creates a matrix of characters, of which we take the last column. Read from the top to the bottom, this last column is the BWT of string s. This algorithm is also illustrated in figure 2.1. More efficient algorithms for creating the BWT of a string exist, which do not require building up the entire matrix of cyclic rotations.

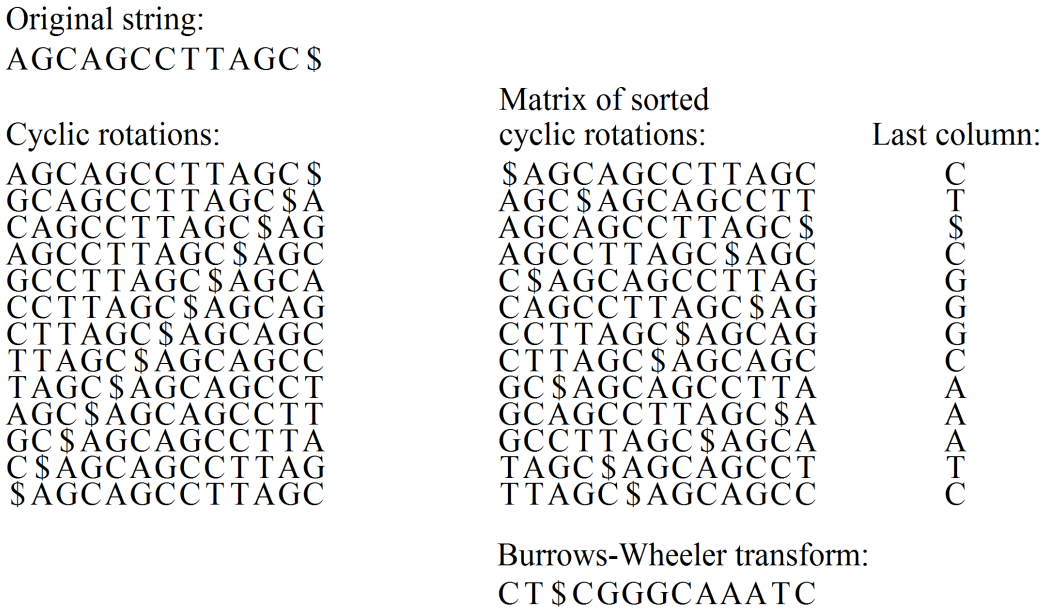

Figure 2.1: Generating the BWT of the string $s =$ AGCAGCCTTAGC$. *After generating a list of cyclic rotations, the list is sorted, and the last column of the matrix formed by this sorted list reveals the BWT of* s *to be* CT$CGGGCAAATC.

It is customary to append a certain character to the end of the string before generating its BWT, with that character being different from all other characters in the string. A usual choice for this character is the dollar sign "$", which is assumed to be lexicographically smaller than all characters in the original string.
The reason for appending such a character is the requirement of being able to reconstruct the original string from the BWT, which allows us to discard the original string after encoding it using the BWT and compressing it with run-length encoding. Without a dedicated character telling us where the end of the string is located, the matrix of alphabetically sorted cyclic rotations could be reconstructed from the BWT, but it would be impossible to decide which one of the rows in that matrix represented the original

string, as cyclic rotations based on each row would alphabetically sorted to the exact same matrix. Instead of adding a distinctly different character to the end of the string, it is therefore also possible, although not customary, to simply store an integer with the BWT containing the number of the row within the matrix which contains the original string (Nelson, 1996).

As mentioned before, it is crucial that the BWT is a reversible reordering of a string, such that the original string can be obtained from the BWT. A simple algorithm for recreating the original string given its BWT starts by reconstructing the alphabetically sorted matrix of cyclic rotations as empty $n \times n$ matrix, where n is the length of the BWT. The last column of the matrix is then filled with the BWT, spelled out vertically. All the rows in the matrix are then sorted alphabetically. Now, the second-to-last column of the matrix is filled with the BWT, and again all the rows in the matrix are sorted alphabetically.
This inserting of the BWT in the right-most empty column and subsequent alphabetical sorting of all rows is continued until the matrix has been filled entirely. The original string now consists of the first $n - 1$ characters in the row in the matrix which ends with the special character, usually the dollar sign.

As noted before, although the BWT originally came into use to aide in compressing strings, we actually use it for a different reason. Ferragina and Manzini (2000) proposed to combine the BWT with a structure known as a suffix array, to form the FM-index, or Full-text index in Minute space. The advantage of doing so is that we get an overall data structure which is not much larger than the string which we encode, but which allows us to perform fast lookup operations on it which enables us to quickly align reads to a reference string encoded in such a way. Therefore, many current alignment tools such as BWA-MEM (Li, 2013) and Bowtie (Langmead et al., 2009) use the BWT to encode the reference and work on it in this compressed form directly.

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 (Myers, 2005).

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 (Li, 2014a).
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. Formal Graph Definition

We define a *graph* $G = (V, E)$ as a collection of a set of *nodes* $V = \{v_1, ..., v_n\}$, where $n = |V|$, and a set of edges $E \subseteq V \times V$. Each one of these edges is a set $(u, v) \in E$, which we call an edge from node $u \in V$ to node $v \in V$. As the graphs we are working with are considered to be *directed*, we assume edge $(u, v) \in E$ to be distinct from edge $(v, u) \in E$. An example of such a graph can be seen in figure 2.2.

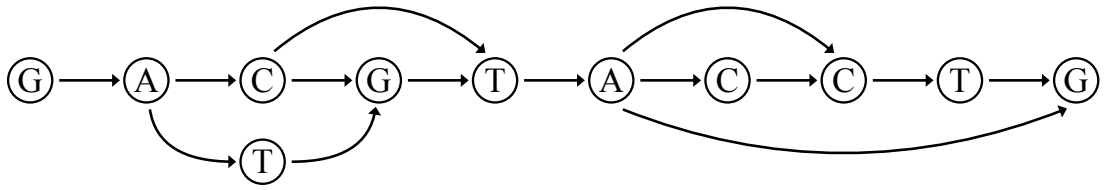

Figure 2.2: Example graph with 11 nodes and 14 edges.

For every node $v \in V$ we define the amount of incoming edges of node v, or the *indegree* $\text{in}(v)$, as the number of distinct edges (u, v) contained in E for any $u \in V$. Similarly, for every node $v \in V$ we also define the amount of outgoing edges of node v, or the *outdegree* $\text{out}(v)$, as the number of distinct edges $(v, u) \in E$ for any $u \in V$.
The graphs considered here are *labelled*, which means that a label $l(v)$ is attached to every node $v \in V$. Every label $l(v)$ represents a character from the alphabet Σ, which is a set of characters. In principle, a label could also consist of several characters, but we limit each label to exactly one character as we wish to represent a word formed by characters from Σ as a sequence of nodes connected by edges.

We refer to such a structure of sequentially connected nodes $u_1, u_2, ..., u_n$ as *path* $P = u_1u_2...u_n$. The label $l(P)$ of this path is then the word $l(u_1)l(u_2)...l(u_n)$ formed by the labels of each node.

2.4. 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 et al. (2002) 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 Mäkinen et al. (2010).
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 (Sirén et al., 2014).
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 (Puglisi et al., 2007) 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 (Navarro and Mäkinen, 2007).
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 (Sirén et al., 2014):
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.

2.5. 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 Huang et al. (2013). It works on the core assumption that most differences 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 (Holt and McMillan, 2014).

2.6. 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 Schneeberger et al. (2009).
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.

2.7. Extending the Burrows—Wheeler Transform

Efforts involving changes to and generalizations of the BWT are also aided by the development of the XBW, or extended Burrows–Wheeler Transform. This is a method of compressing trees similar to how the BWT is used to compress sequential data (Ferragina et al., 2009).
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.

Sirén et al. (2014) 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 encode graphs. In particular, any prefix-sorted reverse-deterministic acyclic connected finite graph can be encoded in this way.
The graph being finite means that it needs to contain a finite amount of nodes and edges. It being connected means that all nodes need to be reachable by all other nodes, when being allowed to follow edges in both ways. Finally, the graph being acyclic means that it is not allowed to contain any cycles, which are paths on which we leave a node, follow one or more directed edges, and arrive at the same node again. The requirements of the graph being reverse-deterministic and prefix sorted will be defined in sections 3.8 and 3.9, respectively. As seen in these sections, these requirements are not actually limiting us in practice, as we can convert any acyclic connected finite graph into a prefix-sorted reverse-deterministic one.
In the paper, the extended XBW is stored in two different kinds of data structures, which have not been given explicit names there. We refer to these structures as XBW node tables and flat XBW tables. Each such table unambiguously encodes exactly one graph, but a given graph can usually be encoded by different tables.
Before a graph can be encoded by an XBW table, we append an *end node* v_e with the label "$" to its end, like we did for encoding a string using the BWT. By appending it to the end we refer to creating an edge from each node with outdegree zero to the new node. We also insert a *start node* v_s with the label "#" at the beginning of the graph, where we assume that the hash tag character is lexicographically bigger than all characters contained in the other node labels. This means that we create an edge from the new node to each node with indegree zero.

2.7.1. Node Tables

We decided to refer to the first of the two data structure as XBW node table, or just node table, as it is a table containing the extended XBW which contains exactly one column for each node of the graph represented by the extended XBW.

More formally, we define:

Definition An *XBW node table* is a table with rows for the BWT, the prefixes, the values of the bit vector M, and optionally the values of the bit vector F. These rows together represent an acyclic 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 i representing node v_i in the graph, the value of the BWT in column i represents the labels of the nodes preceding v_i. If several nodes with different labels are preceding the node v_i, 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 2.3 which is represented by node table 2.2.

The value of the prefix in column i is precisely the prefix of node v_i in the graph, which

Table 2.2: Node table corresponding to the graph in figure 2.3.

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

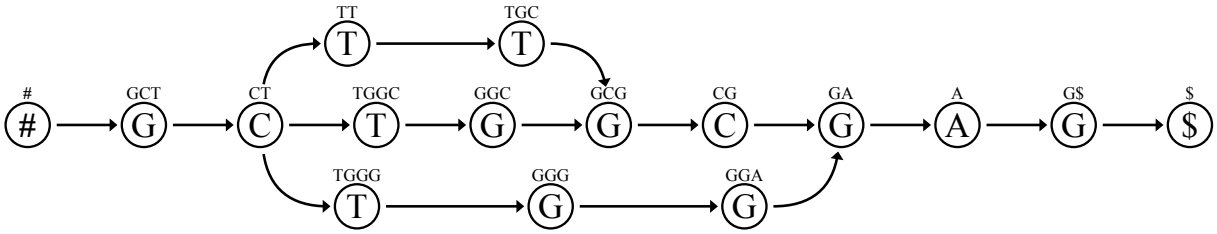

Figure 2.3: Graph corresponding to node table 2.2. The prefixes of the nodes have been provided in small print above the nodes, to make it easier to identify which node in the graph corresponds to which column in the table.

is a string starting with the label of v_i and then containing the labels of the following nodes in sequence of encountering them traversing the graph outwards from v_i. The value of the cell M in column i is a sequence of bits whose length corresponds to the amount of successor nodes of v_i, or equally the amount of outgoing edges of node v_i. This sequence always starts with the bit "1" and is then followed by as many "0" bits as are necessary to achieve the desired length. In addition to bit vector M, a second bit vector designed as F can be used within a node table. The value of F in column i is a bit sequence whose length corresponds to the amount of predecessor nodes of node v_i. The reason why using the F bit vector is not strictly necessary is that its information for any column i is already encoded in the length of the BWT cell of column i. When implementing M and F, it is

possible to think of them as arrays of strings, with each string containing the characters "1" and "0", for ease of implementation.

2.7.2. Flat Tables

In contrast to the previously introduced node table, we refer to the next data structure as flat XBW table, or just flat table. This name is inspired by the fact that the data of each row is not nested in an array with elements of various lengths, but instead stored as a flat sequence of characters which can be encoded as a string, or as an array with elements of fixed size.
The definition reveals that despite their differences, flat tables are nevertheless similar to node tables:

Definition A *flat XBW table* is a table with rows for the BWT, the node labels and the values of M and F, representing an acyclic finite graph. The table is arranged such that each cell contains exactly one character for the BWT and the node labels, and such that each cell contains exactly one bit for the bit vectors M and F.

We decided to refer to the row containing the node labels as first column row, or short "FC". The reasoning behind this name is that the row of node labels of the graph relates to the BWT row, as the first column in the matrix of sorted cyclic rotations does to the last row, which is the traditional BWT. That is, the node labels are also the alphabetically sorted BWT, and in fact the columns are sorted in such a way that when the prefixes are generated for each column in turn, they are sorted alphabetically as well. This means that the FC row can easily be calculated by sorting the BWT row alphabetically.
In addition to the FC row we usually calculate three further data structures based on the BWT, M and F rows. We call these "Σ", "ord" and "C".
Σ is the alphabet of the graph, which is a set of all distinct characters encountered in the BWT.
The ord array has the characters of the alphabet as keys and their positions in the alphabetically sorted Σ as indices.
The array C also has the characters of the alphabet as keys. Its values are the indices of the first occurrences of these characters within FC. Using C, we can completely reconstruct FC, and accessing an arbitrary cell in the FC row via an iteration over C is a very quick operation. Therefore, we do not need to explicitly store FC in any other way. Of course, we can do so anyway for a particularly simple implementation.
Σ, ord and C, which represents FC, are all very small data structures and do not take up a lot of space. We therefore usually keep them pre-calculated in memory, instead of generating them on the fly based on the BWT, M and F again and again.

The bit vectors M and F are exactly the same as the corresponding bit vectors in the node table, and are each strongly related to one of the other rows. In particular, each cell in the M vector corresponds to a cell, or character, in the FC row, as each outgoing edge (u, v_i) of a node u corresponds to another cell in the M vector, and each of these

outgoing edges shares the same node u in which it originates, therefore sharing the same label $l(u)$, which is precisely the content of the FC row.
On the other hand, each cell in the F vector corresponds to a cell, or character, in the BWT. The reason for this is that the graph we are working on is necessarily reverse deterministic for this encoding to function properly, meaning that each node has no two predecessors with the same label. This is explored in further depth in section 3.8. Due to that, the label $l(u_i)$ for each incoming edge (u_i, v) into a node v is distinct, as the graph would otherwise not be reverse deterministic. Therefore, for every node v there are as many entries in the BWT, which contains the labels of the preceding nodes, as there are entries in the F bit vector, which keeps track of the amount of incoming edges.

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. We therefore should not even think about whole columns in the flat table, but instead of clusters of cells which correspond to nodes. We use the M and F rows to find these cell clusters across different rows, which for graphs with high in- and outdegrees are not necessarily located in columns that are even close together. How cells in a flat table relate to actual nodes in a graph is illustrated via specifically highlighted cells in table 2.3, which corresponds to the graph with highlighted nodes in figure 2.4.
More formally, we can observe that if we have a cell in column i of the FC row, and

Table 2.3: A flat XBW table with cells highlighted that correspond to four nodes in the graph in figure 2.4 represented by this table. The alphabet for this table, including the labels of the start and end node, is $\Sigma = \{\$, A, C, G, T, \#\}$. *The* ord *array is* $[\$ \Rightarrow 0, A \Rightarrow 1, C \Rightarrow 2, G \Rightarrow 3, T \Rightarrow 4, \# \Rightarrow 5]$ *and the* C *array is* $[\$ \Rightarrow 0, A \Rightarrow 1, C \Rightarrow 2, G \Rightarrow 6, T \Rightarrow 13, \# \Rightarrow 17]$.

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

want to find the other cells corresponding to the node which this cell belongs to, then we first of all check the value of M in column i. If it is zero, we go to the left until we reach a column j which contains a one in M. If it is not zero, then we set $j = i$. We now go to

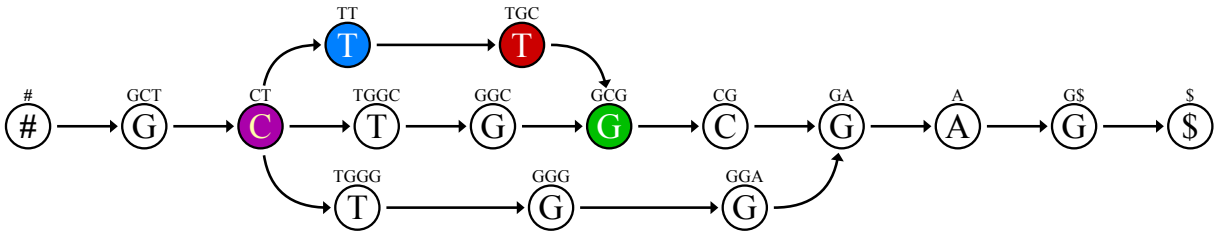

Figure 2.4: Graph corresponding to flat table 2.3. The nodes highlighted in this graph correspond to the cells highlighted in the flat table.

the right from cell $j+1$ onwards, until we encounter a one in M in column k. This shows us that all the FC values and all the M value from columns j until $k-1$, including both j and $k-1$, belong to the same node in the graph.
To find the corresponding BWT and F values, we iterate over all columns of the table from the beginning on the left until we reach j, and count the occurrences of ones in M. We refer to this operation as computing the *rank* of character 1 in sequence M until the limit j, which we denote as $\text{rank}_1(M, j)$ (Sirén, 2009). We then iterate over the columns of the table again from the beginning on the left, increasing a counter every time that the bit vector F contains a one, until we reach the result that the rank operation gave us. We refer to this as a *select* operation, selecting character 1 in sequence F until the limit l, where l in this case is set as the result of the rank operation before. In general, we denote such a select operation as $\text{select}_1(F, l)$ (Sirén, 2009).
The result of the select operation is now the index h of the column in which the BWT and F values of the node we are interested in start. We again go on to the right through F until we encounter the next one in column g, and finally know that the node which we were interested in corresponds to the FC and M values in columns j until $k-1$ and to the BWT and F values in columns h until $g-1$.
Similar operations can be performed to find the range in FC and M given an index into the BWT, namely ranking the index in F and calling a select operation on M to find the corresponding start.

The previous approach assumed that we were given the index i of a cell in the FC row, and were interested in finding all other cells corresponding to the same node as that cell. We also mentioned a second approach based on an index of a cell in the BWT row. However, we can lastly introduce a more general indexing approach independent of whether we are considering the FC row or the BWT row.
The underlying insight for this third indexing method is that although the cells in the BWT and FC rows corresponding to a node can lie far apart, the order of nodes in both rows is exactly the same. More formally, if the cells corresponding to node u are sorted before the cells corresponding to node v in the FC row, then the cells corresponding to node u are also sorted before the cells corresponding to node v in the BWT, and vice versa. The reason for this to be true is that the shift in cell locations occurs due to zeroes in the M and F vectors, which can expand the width of an area taken up by a node, but cannot change the order in which the nodes are sorted.
Therefore, a viable method for indexing a node v in a flat table is to simply give an integer i corresponding to how many nodes u_j are encoded before, or to the left of, node v. To

find the cells in the FC row corresponding to node v we can then call $\text{select}_1(F, i)$ to find the first column and append columns while F is zero. To find the cells in the BWT row corresponding to node v we can call $\text{select}_1(M, i)$ to find the first column and append columns while M is zero.

Overall we now have three ways to address a node within a flat XBW table: We can use an integer index corresponding to a column in the FC row, which we will refer to as *FC-indexing*, we can use an integer index corresponding to a column in the BWT row, which we will refer to as *BWT-indexing*, and we can use an integer index corresponding to the position of a node in the table, which we will refer to as *absolute indexing*.
As an example, we can consider the node highlighted in green in table 2.3. Its FC-index is 8, as we start to count from 0 and the first green column in the FC row is in position 8. Its BWT-index is 7, as the first green column in the BWT row is in position 7. Finally, its absolute index is 6, as there are 6 nodes preceding it to the left, one with label "$", one with label "A", two with label "C" and two with label "G".
Calling $\text{select}_1(M, 6) = 8$ gives us the FC-index from the absolute index and calling $\text{select}_1(F, 6) = 7$ gives us the BWT-index from the absolute index. We can also call $\text{rank}_1(M, 8) = 6$ to get the absolute index from the FC-index and $\text{rank}_1(F, 7) = 6$ to get the absolute index from the BWT-index. Conversions from the FC-index to the BWT-index and vice versa can therefore simply be executed as conversions to the absolute index and back.
To find the FC-index at which the cells of the node end, we can call $\text{select}_1(M, 6{+}1){-}1 = 8$, which is the same as the FC-index at which the cells of the node start, as the node only occupies one column in the FC row. To find the BWT-index at which the cells of the node end, we can call $\text{select}_1(F, 6 + 1) - 1 = 8$, which is one more than the BWT-index at which the cells of the node start, as the node occupies two columns in the BWT row.

Each of the three indexing methods presented here may be used in practice, and choosing one of them as standard to which we always need to conform would create unnecessary computational effort for conversions between them. Therefore, whenever work within flat XBW tables is performed, great care must be taken to make clear which kind of indexing is used at each step.

We conclude the explanation of node tables and flat tables by remarking that while each column in an XBW node table corresponds to a node in the graph, each column in the flat table can be thought of as corresponding to an edge in the graph, if the BWT and F rows are thought of separately from the FC and M rows. That is, each of the columns in the BWT and F rows corresponds to an incoming edge for some node, and each of the columns in the FC and M rows corresponds to an outgoing edge for some node. Therefore, we are always assured that the total length of the BWT, F, FC and M rows are the same within the flat table, as the total amount of outgoing edges from all nodes in the graph is the same as the total amount of incoming edges into all nodes in the graph.

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. 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++ (Taivalsaari et al., 2008). 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 and compared existing data formats for genomic graphs as well as designed several new data formats to be able to better understand the strengths and weaknesses of 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 (Childs, 2007). Its simple structure as a plain text format consisting of a definition line followed by the genomic data 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. A FASTA-like format designed to handle graph data is FASTG (Jaffe et al., 2012).
This format was especially designed to enable quick conversion from a file encoded in FASTG back to FASTA to enable backwards compatibility with tools which cannot handle the graph portion of the data. In that case, all the graph portions are simply left out and a path through the graph becomes the genomic string encoded in the converted FASTA file.
In the FASTG format, genomic graphs are represented by providing the global structure of the data, which describes the overall structure of the entire graph, into which linear sequences and local graphs are embedded. All of this is then surrounded by specific begin and end lines which identify the file as being in the FASTG format. An example for a graph encoded in the FASTG format can be seen in figure 3.1.

```
#FASTG:begin;
#FASTG:version=1.0:assembly_name='example graph';
>chr1;
GA[1:alt:allele|C,T]GT
>chr2:chr2;
AAGT
#FASTG:end;
```

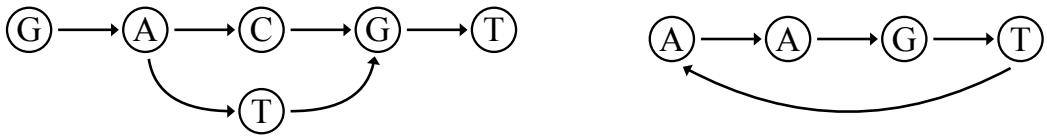

Figure 3.1: FASTG example file together with the graph that the file represents.

The global structure is hereby represented by several data blocks, each of which contains a definition line starting with the character ">", followed by one or several lines representing the genomic data within that block. The text in the definition line hereby determines how several such data blocks relate to each other and the global structure of the encoded data, such that a data block can represent a cycle or a linear part, and its ends can be attached to other data blocks. They however do not have to be attached to other data blocks, such that it is also possible to encode graphs in the FASTG format which are not connected, meaning that not all nodes within the graph need to be reachable from all other nodes.
The linear structure of the data is encoded via special commands right inside the data block. Examples for some common local commands are shown in table 3.1.

When referring to "default" values in this table, we mean that these are the values taken when the FASTG file is converted to FASTA and not the entire variety of the data can be upheld.

Table 3.1: Examples for local commands in FASTG.

Example	**Remarks**
`[6:gap:size=(6,4..8)]`	Gap of four to eight characters, with default gap length of six characters.
`[1:alt|C,CG,TT]`	Bubble representing the options of "C", "CG" or "CT" with the default option being the first one.
`[1:alt:allele|A,T]`	Bubble representing the options of "A" or "T" with the default option being the first one. We use the `allele` term to denote that we are sure that both options occur in the sample data.
`[10:tandem:size=(5,3..7)|CG]`	Three to seven repeats of "CG", with the default being five repeats.
`[20:digraph:...]`	A local graph within the format which takes up a default length of twenty characters. The actual graph definition inside the `digraph` command is replaced by "`...`" for brevity.

3.1.2. GFA Format

The GFA or Graphical Fragment Assembly format has been proposed in 2014 (Li, 2014b,c). Despite some initial criticism (Knight, 2014), this young format has a lot of advantages over FASTG. Such advantages are that it is more flexible, due to encoding graphs of any arbitrary size without making a distinction between local and global graphs, and that it can easier be read by programs, as not as much emphasis is put on making it human-readable. Therefore, the focus can lie clearly on the needs of developers of advanced software packages, and on allowing them to directly store the kind of data that they encounter.

A file in the GFA format consists of several lines of tab-delimited plain text, possibly including optional tags. Each line hereby starts with a character which indicates what kind of a line it is (The GFA Format Specification Working Group, 2015). Figure 3.2 contains an example graph encoded in the GFA format.

The lines of the file header start with the character "`H`" and can be followed by tags which tell us about the version number or the quality of the data. Other line types include ones with character "`S`" representing sequences, which contain the name of the sequence and the genomic sequence itself, lines with character "`C`" for containments and lines with character "`L`". These represent links between sequences which are identified by their names and orientations. The links are augmented by CIGAR strings. In general, such a CIGAR string is a sequence of lengths and operations associated with these lengths that is used when aligning one string to another (The SAM/BAM Format Specification Working Group, 2015). Commonly, such a string can 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 (Li,

```
H   VN:Z:1.0
S   1   ACA
S   2   AAG
L   1   +   2   +   1M
S   3   AGGT
L   2   +   3   +   2M
S   4   TG
L   3   +   4   +   1M
S   5   TCC
L   4   +   5   +   0M
L   2   +   5   +   0M
S   6   CACGT
L   1   +   6   +   2M
L   6   +   4   +   1M
```

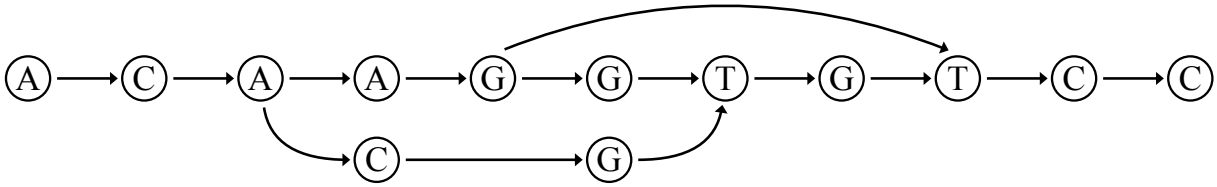

Figure 3.2: GFA example file together with the graph that the file represents.

2014b; Li et al., 2009).

3.1.3. Bubble Format

Having explored the FASTG and GFA format, we decided to also look at the bubble notation which is already partially in use within other formats, and base a data format solely on it. In other data formats, such as FASTG, bubbles are included as a way to encode very short graphs without the explicit generation of nodes and edges for them, as that would lead to a lot of overhead and decrease human-readability. On the other hand, adding such bubbles as options to formats which otherwise would already be able to contain said graphs through more complicated encoding mechanisms does make it more complicated for programs to work with that format, which is why the inclusion of such bubbles is not always seen positively (Li, 2014b).
Our format based on these bubbles 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 have a look into this particular format, as implementing it in a software package 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 this format 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 3.3.

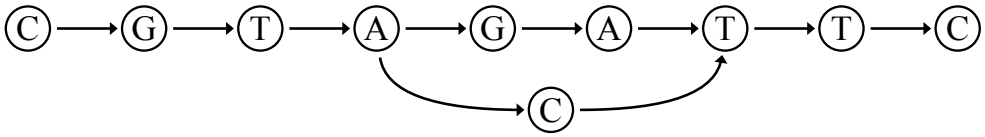

Figure 3.3: Simple bubble, represented in bubble format as CGTA(GA|C)TTC.

Insertions and deletions are represented as bubbles whose alternative route is empty. One such deletion can be seen in figure 3.4.

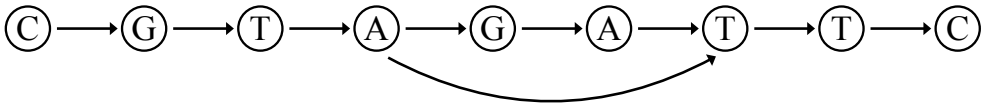

Figure 3.4: Deletion, represented in bubble format as CGTA(GA|)TTC.

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 3.5. Nevertheless, the bubble format is sufficient for simple algorithmic tests.

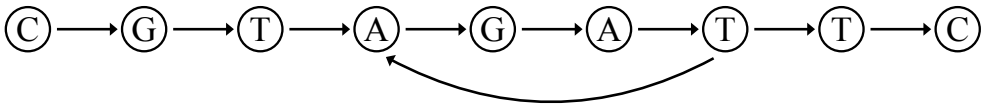

Figure 3.5: Cycle which cannot be fully represented in bubble format, but can be approximated to any wanted length as CGTAGAT(|AGAT(|AGAT(|AGAT(|...))))TC.

3.1.4. GML Format

After implementing the bubble format, which represents a very straightforward but rather limited approach for encoding graphs, we decided to design a different new format to encode genomic graphs in the Graph Merging Library. 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, while not falling into the pitfalls of the bubble format and limiting the scope of the graphs which could be expressed too much. 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.
An example graph encoded in the GML format can be seen in figure 3.6.

An aim when designing the GML format was to not unnecessarily complicate the process of making existing tools of the analysis pipeline compatible with graph data. Therefore,

```
>gml_example_graph
GAGTCGAT|p1,1,TGG,5;,p1:1,T,mp:6
```

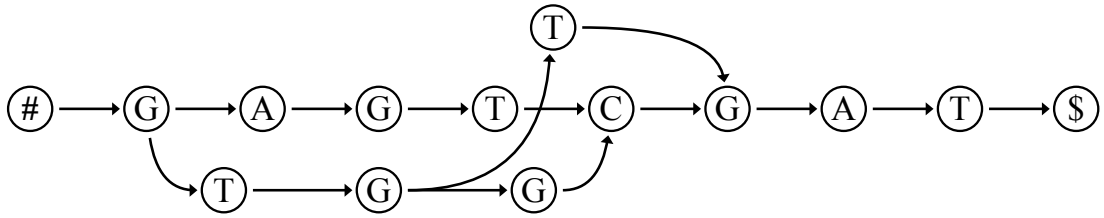

Figure 3.6: GML example file together with the graph that the file represents.

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.
An example for how several graphs can be encoded in the GML format can be seen in figure 3.7.

```
>first_gml_example_graph
CAA|,1,CG,3
>second_gml_example_graph
TGTC|,1,,4;,1,,3
```

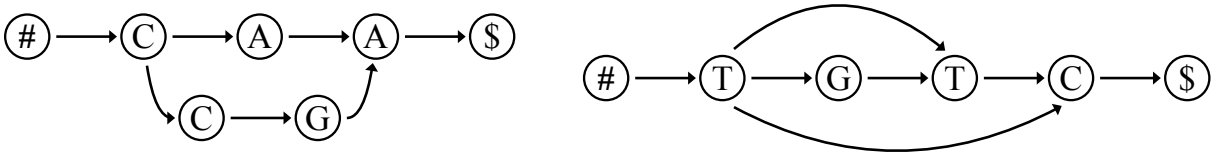

Figure 3.7: GML example file containing two graphs together with the graphs that the file represents.

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*, which has no incoming edges, to the *end node*, which has no outgoing edges. 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 path, 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 section 3.16, 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. An FFX example file and the graph that it represents can be seen in figure 3.8.

The basic structure of an FFX file is again inspired by FASTA, containing an optional

```
>ffx_example_graph
B   CGTAG^GAGAAATC$
M   111111111001111
F   111011111011111
O   $>0,A>1,C>2,G>3,T>4,^>5
C   $>0,A>1,C>6,G>8,T>12,^>14
```

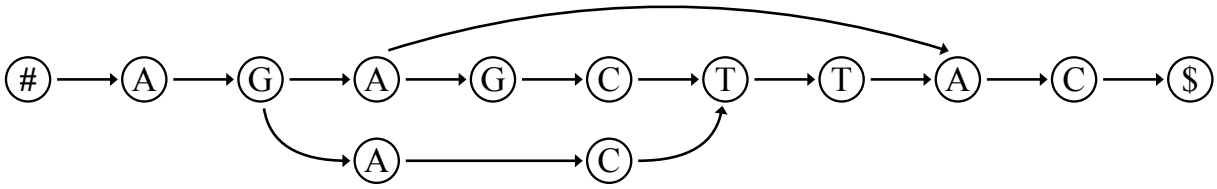

Figure 3.8: FFX example file together with the graph that is represented by the file. This file contains the character "^" instead of "#", as we use this one internally due to its lexicographic behaviour as explained on page 32.

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. For a more detailed explanation of how fused flat tables work and the motivation behind using them, see section 3.16.
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, these fused data structures are required to be strictly linear. This means that a data block is always fused to the one directly following within the file. Just like the encoding of the global structure happens in FASTG, 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 as complex a graph as is necessary as long as it does not contain any loops. Also, this is inspired by the actual needs for a huge graph reference, in which the overall structure is linear and all perturbations are on a rather local scale.
Figure 3.9 contains an example for how several graphs can be encoded in the FFX format as a single fused graph.

```
>first_fused_graph
B   AAGC^C$
M   1111011
F   1101111
O   $>0,A>1,C>2,G>3,^>4
C   $>0,A>1,C>3,G>5,^>6
>second_fused_graph
B   AGTT^$
M   111101
F   110111
O   $>0,A>1,G>2,T>3,^>4
C   $>0,A>1,G>2,T>3,^>5
```

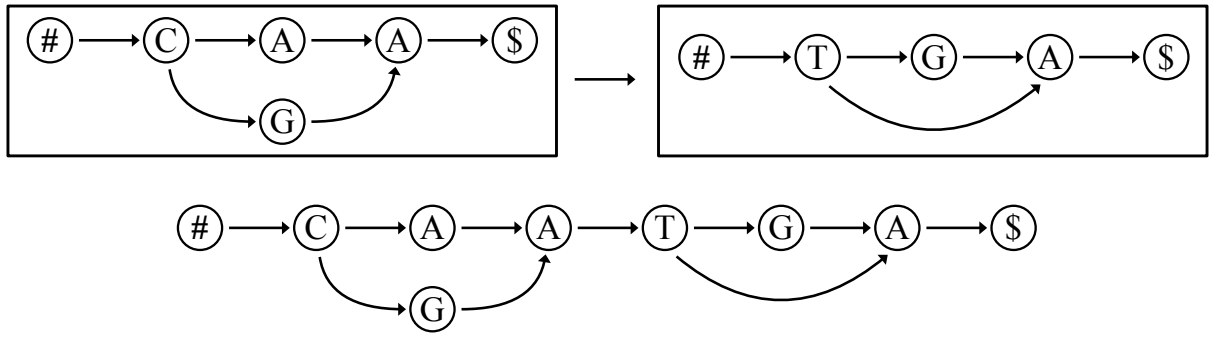

Figure 3.9: FFX example file containing two graphs (top) together with the fused graph that is represented by the file (middle) and the graph that the fused graph represents (bottom).

In FFX, each data block consists of lines for the BWT, the M and F bit vectors, as well as lines for the ord and C arrays. Both of these arrays could be constructed on the fly from just the BWT. This however takes up a lot of time while they both only take up a small amount of space even for very large graphs, such that storing them explicitly within the file means that the lengthy process of computing them does not need to be repeated again and again whenever the file is opened.
In particular, the BWT is contained in a line starting with the letter "`B`" followed by a tab character. Similarly, the bit vector M is contained in a line starting with "`M`" and the bit vector F in a line starting with "`F`". Finally, the ord and C arrays are contained in lines starting with "`O`" and "`C`", respectively.

3.1.6. Implementation

To compare the different data formats that have been proposed, we wrote several scripts in Python. These scripts support 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 results gained by implementing the different data formats.

3.2. 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

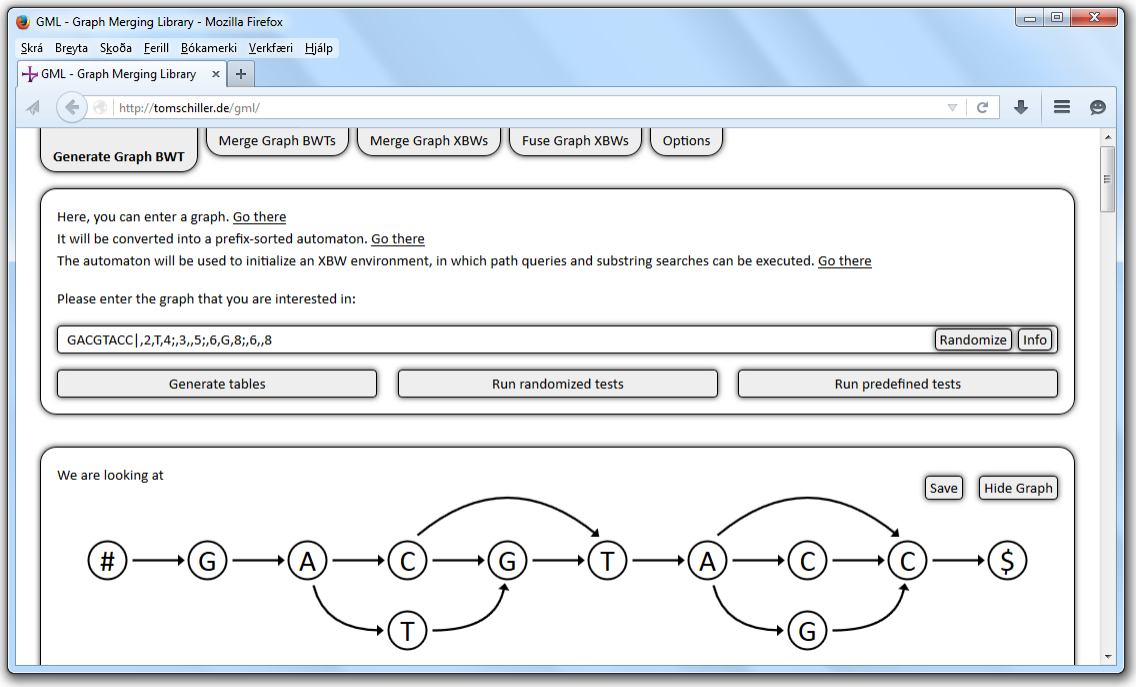

Figure 3.10: Graphical User Interface of the Graph Merging Library.

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 3.10.
In particular, GML includes different methods for merging and fusing genomic graphs, which may be helpful to other future projects.

3.3. 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. In particular, graphs considered in GML are finite automata.

3.3.1. Graph Properties

A graph $G = (V, E)$ used in the Graph Merging Library is generally considered to consist of labelled nodes and directed edges, as introduced in section 2.3.
We assume that the graph is *connected*. We define a graph to be connected if every node

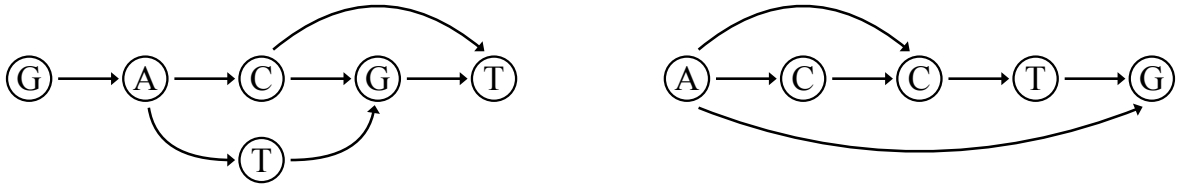

Figure 3.11: A graph which is not connected, consisting of two clusters of nodes. The nodes within each cluster are connected among each other through edges, but are not connected to nodes of the other cluster.

of the graph is connected to every other node of the graph through some edges, even if not every node is 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. An example for a graph violating this assumption can be seen in figure 3.11.
Also, the edges in the graph need to be *unique*. This means that no two directed edges

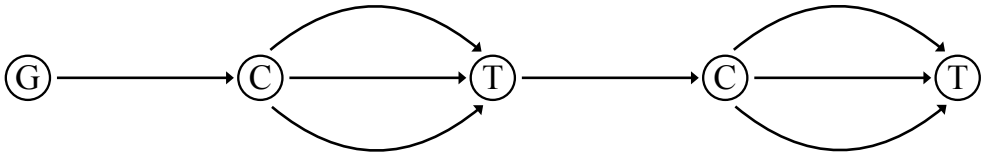

Figure 3.12: A graph with edges which are not all unique, as there are several edges which share the same origin and target node.

$(u_1, v_1) \in E$ and $(u_2, v_2) \in E$ are allowed to exist within the same graph for which both $u_1 = u_2$ and $v_1 = v_2$ are true. A graph not obliging with this requirement is shown in figure 3.12.
Graphs within the Graph Merging Library are supposed to be *acyclic*, meaning that they are not allowed to contain any loops. These are paths along the directed edges which lead from a node onto itself, either directly or along a sequence of edges through other nodes.
The alphabet Σ is assumed to contain only upper case English letters for graphs within GML.

Graphs are also assumed to have exactly one *start node* v_s with the label "#" which is the start of each path traversing the entire 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. The lexicographic value of the character "#" is supposed to be above the lexicographic value of all characters in the alphabet Σ, which is however not the case for "#" in ASCII (Smith, 1967). We therefore internally use the character "^", whose lexicographic value encoded in ASCII is above all English letters in upper case. This is also the reason why we work with upper case letters as node labels instead of allowing lower case letters as well. Before being displayed, the character "^" is replaced by "#".
Furthermore, each graph is assumed to have exactly one *end node* v_e with the label "$" in which each path traversing the entire graph ends. This can be reformulated as there being only one node without any outgoing edges, this node having the label "$", and no other node in the graph having this label. The lexicographic value of the character "$" is supposed to be below the lexicographic value of all characters in the alphabet Σ, which is the case for "$" if the alphabet only contains English letters and the encoding in which string characters are compared is ASCII.
It should be noted that we use two different conventions concerning an edge from the end node v_e to the start node v_s. When we are working with XBW node tables or with flat XBW tables, we assume that there is an edge leading from the end node to the start node, but that there still is no other incoming edge into the start node and no other outgoing edge out of the end node. This simplifies working with these tables slightly, as the FC row then exactly corresponds to the alphabetically sorted BWT row. If there was no such edge from v_e to v_s, then the BWT would contain the symbol "#", but it would be missing the "$" character. Also, the FC row would contain the "$" character, but it would be missing the "#".
On the other hand, when we are considering graphs internally as arrays of nodes which each contain a label, an array of predecessors and an array of successors, then we explicitly assume there not to be an edge from the end node to the start node. This means that the predecessor array of the start node is empty, and the successor array of the end node is empty, which can be checked as alternative to reading out the label of the node and checking whether it is a special character.

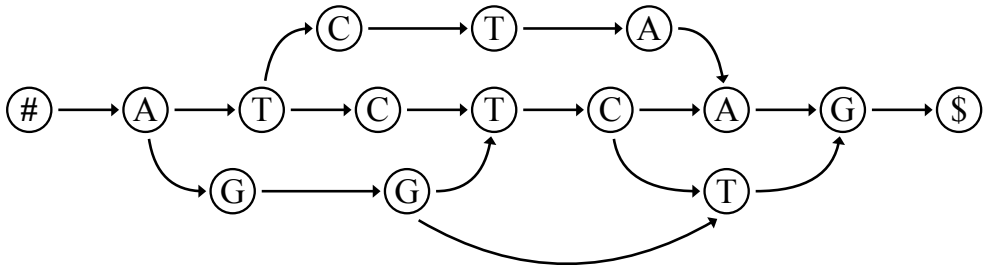

Figure 3.13: A graph fulfilling the core assumptions of GML.

A graph which is not violating any of aforementioned assumptions can be seen in figure 3.13.

3.3.2. Additional Assumptions for Merging Graphs

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.

In addition to this extra requirement, we also take care with the end node of the first graph to be merged and the start node of the second graph, as these are special nodes which will be deleted after the merging has been finished.
Just as with "#", which encoded in ASCII does not have the lexicographical properties that we require, we also replace two other special characters internally with different characters. These are "$\$_0$" and "$\#_0$", which we use while merging graphs. In particular, we sometimes replace the label of the end node of the first input graph with "$\$_0$" and the label of the start node of the second input graph with "$\#_0$", to better be able to distinguish them from the start and end nodes of the merged graph. However, we need to encode them as single characters when working with flat tables which do not allow some cells to have different lengths. Therefore, we internally encode "$\$_0$" as the percent sign "%" and we internally encode "$\#_0$" as the underscore sign "_".
A clear lineup of all special characters which we use in GML can be found in table 3.2.

Table 3.2: Special characters in GML together with information about how they are encoded internally as well as remarks about their use.

Character	**Encoded as**	**Remarks**
$	$	label of the end node v_e
#	^	label of the start node v_s
$\$_0$	%	label of the end node of the first input graph
$\#_0$	_	label of the start node of the second input graph
!	!	error character during prefix generation

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, a valid GML data block can be entered directly into an input field, or a graph can be created randomly for testing purposes.

3.4.1. Randomly Generating New Graphs

The Graph Merging Library contains the function `generateRandomGraphString` for the automatic generation of random graphs, which can be used to quickly generate input for testing the provided merging functions. This function can be called without any arguments, in which case it will generate short graphs or strings. If necessary, the length of the main path can however also be provided to generate graphs of different lengths. The alphabet can also be specified, which otherwise will be set to $\Sigma = \{A, C, G, T\}$. Finally, two thresholds with values between 0% and 100% can be specified, which tell the function how likely it is to add at least one info block to the graph, and how likely it is to add more info blocks after adding the first.

3.4.2. Internal Graph Format

When a GML file is being opened, then some work still needs to be undertaken to convert the opened graph into the extended XBW format, and the graph is stored in a different format internally at first before this conversion occurs.
That is, each graph is stored as array which contains all nodes as elements. Every node here is an object which has the properties `c`, `n`, `p` and possibly `f`. The `c` property contains one character, which is the label of the node. The `f` property, if it is defined, contains the prefix of the node. Until the prefixes have been calculated, these properties can be missing.
The `n` property contains an array of integers, with each integer being the index of a node in the array representing the graph. It represents a list of all succeeding nodes. Similarly, the `p` property is a list of preceding nodes, implemented as array of integers, with each integer again being the index of a preceding node in the graph array.
The start node with label "#" is assumed to be found as the first element of the graph array. Apart from this node, the positions of the other nodes within the array are not significant. In fact, elements of the array are also allowed to be empty or the boolean value `false`. This is helpful when deleting nodes from the graph, as the nodes with higher indices therefore do not need to be re-indexed, which would entail a lot of work as then all the arrays of preceding and succeeding nodes also would need to be updated accordingly. Therefore, every loop iterating over the nodes of the graph needs to check for the current node to actually exist before performing any work on it.

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, but these nodes are not explicitly encoded in the GML format. Therefore, the extra node with the # label is created first when opening a GML file. The main path as specified in the GML file is then added as one possible way of traversing the graph from the # node to the $ node, which is added at the end of the main path.
Finally, the info blocks are iterated over, and their contents are added as edges and nodes which are part of paths off the main path.

3.5. Sanitizing the Graph

If graphs which are used as inputs for the GML merging conform to the assumptions stated in section 3.3, then they can be worked on and results can be given out. However, in case of inputs not conforming to these assumptions, the results of the GML algorithms are not useful or even no results are given out, as infinite loops are encountered. Therefore, a sanitization step is used to ensure that the input actually conforms to the stated assumptions. If this is not the case, which can happen for randomly created input graphs, then a warning is given out telling the user that invalid input was encountered and no attempt is being made to actually perform work on the graph.

To perform this check of the core assumptions, the function `sanitizeAutomaton` is called within GML whenever an input graph string is newly converted to an internal automaton data structure representing it. It in turn calls several functions which check one assumption at a time.
The first one of these functions is called `eliminateMultipleEdgesInAutomaton`. Its underlying functionality can be found in algorithm 3.1. It ensures that each edge is unique,

Algorithm 3.1 Unify multiple edges in a graph.

```
1:  for all graph.nodes as node do
2:      if length of node.predecessors > 1 then          ▷ Unify array of preceding nodes
3:          new_array ← [ ]
4:          for all node.predecessors as predecessor do
5:              if not ( new_array contains predecessor ) then
6:                  append predecessor to new_array
7:              end if
8:          end for
9:          node.predecessors ← new_array
10:     end if
11:     if length of node.successors > 1 then          ▷ Unify array of succeeding nodes
12:         new_array ← [ ]
13:         for all node.successors as successor do
14:             if not ( new_array contains successor ) then
15:                 append successor to new_array
16:             end if
17:         end for
18:         node.successors ← new_array
19:     end if
20: end for
```

meaning that there is no pair of edges $(u_1, v_1) \in E$ and $(u_2, v_2) \in E$ for which both $u_1 = u_2$ and $v_1 = v_2$ are true. It does not actually throw an error if such edges are encountered, but simply deletes one edge of the pair, until none of these pairs of edges are left.
The next function is `checkAutomatonForCycles`. Its general approach is shown in algorithm 3.2. This function traverses all paths through the graph, starting from the start

Algorithm 3.2 Check if a graph contains cycles.

```
 1: paths ← [[0]]
 2: while length of paths > 0 do
 3:     for all paths as path do
 4:         current_node ← last element of path
 5:         if length of current_node.successors > 0 then
 6:             for all current_node.successors as successor do
 7:                 if path contains successor then
 8:                     return true                    ▷ Same node encountered twice
 9:                 end if
10:                 if last iteration within this for loop then
11:                     append successor to path
12:                 else
13:                     next_path ← path
14:                     append successor to next_path
15:                     append next_path to paths
16:                 end if
17:             end for
18:         else
19:             delete path from paths          ▷ Drop the path if $ node was reached
20:         end if
21:     end for
22: end while
23: return false
```

node, and checks for each new node added to a path if that node has already been traversed within the path before. If such a node is encountered which is traversed several times within one path, then an error is given out and the invalid input warning is displayed to the user.
Finally, the function `checkAutomatonForIncorrectChars` as shown in automaton 3.3 iterates over all nodes of the automaton and checks if the labels conform to the assumptions,

Algorithm 3.3 Check if graph labels contain invalid characters.

```
1: for all graph.nodes as node do
2:     if ( node.label = $ ) and ( length of node.successors = 0 ) then
3:         continue
4:     end if
5:     if ( node.label = # ) and ( length of node.predecessors = 0 ) then
6:         continue
7:     end if
8:     if ( node.label in [A to Z] ) and ( length of node.label = 1 ) then
9:         continue
10:    end if
11:    return true
12: end for
13: return false
```

meaning that every node needs to have a label with exactly one English upper case letter, except for the start node which has the label "#" and the end node which has the label "$". Again, if a node is encountered which does not conform to these assumptions, then the invalid input message is shown to the user and the computation is stopped.
We could also use some function to check if the graph is connected, but this is actually not necessary, as the GML format in which we open the graph can only encode connected graphs anyway. The reason for this is that a graph encoded in GML is encoded as a main path, together with other paths originating and terminating on the already existing paths. Therefore, no path can be encoded which is separate from the others, and checking afterwards if the emerging graph is connected becomes unnecessary.

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 3.4 shows how this function works in general. The basic approach of this algorithm is inspired by the function `automatonToGMLdata`, which we use to convert an automaton into a GML data block.

Algorithm 3.4 Visualize a graph by first displaying one path from the start node to the end node, referred to as main path, and then adding alternative paths around the

established core.

```
1: further_edges ← [ ]                                   ▷ List of edges left
2: done_edges ← [ ]                                      ▷ List of already displayed edges
3: auto_to_path ← [ ]                                    ▷ Map automaton nodes to visualized paths
4: vis_positions ← [ ]                                   ▷ Map automaton nodes to screen locations
5: i ← 0
6: k ← 0
7:
8: while true do                                         ▷ Display main path
9:     for j from 1 to length of auto[i].n do            ▷ Put all edges out of
10:        append [i, auto[i].n[j]] to further_edges     ▷ main path on to-do-list
11:    end for
12:    auto_to_path[i] ← ['mp', k]
13:    k ← k + 1
14:    draw node i on main path (with incoming edge if i > 0)
15:    update vis_positions with position node i
16:    if auto[i].c = '$' then
17:        break
18:    end if
19:    append [i, auto[i].n[0] to done_edges
20:    i ← auto[i].n[0]
21: end while
22:
23: for i from 0 to length of further_edges do           ▷ Display direct edges from
24:    current_edge ← further_edges[i]                   ▷ main path to main path
25:    if current_edge starts and ends on main path then
26:        append current_edge to done_edges
27:        remove i from further_edges
28:        i ← i − 1
29:        draw current_edge
30:    end if
31: end for
32:
33: for i from 0 to length of further_edges do           ▷ Add all remaining
34:    current_edge ← further_edges[i]                   ▷ edges to further_edges
35:    for j from 0 to length of auto[current_edge[1]].n do
36:        if it is neither contained in done_edges nor in further_edges then
37:            append [current_edge[1], auto[current_edge[1]].n[j]] to further_edges
38:        end if
39:    end for
40: end for
41:
42: while length of further_edges > 0 do                 ▷ Draw non-direct paths
43:    for i from 0 to length of further_edges do        ▷ from main path to main path
44:        current_edge ← further_edges[i]
45:        from_n ← auto_to_path[current_edge[0]]
```

```
46:         if from_n is defined then
47:             to_n ← auto_to_path[current_edge[1]]
48:             k ← 0
49:             path_name ← unique name for this path
50:             current_path ← [current_edge[0]]
51:             while to_n is not defined do
52:                 auto_to_path[current_edge[1]] ← [path_name, k]
53:                 k ← k + 1
54:                 append current_edge[1] to current_path
55:                 append current_edge to done_edges
56:                 remove i from further_edges
57:                 for j from 0 to length of further_edges do
58:                     if further_edges[j][0] = current_edge[1] then
59:                         i ← j
60:                         current_edge ← further_edges[j]
61:                         to_n ← auto_to_path[current_edge[1]]
62:                         break
63:                     end if
64:                 end for
65:             end while
66:             append current_edge[1] to current_path
67:             if current_path contains two nodes then
68:                 draw edge between both nodes in current_path
69:             else
70:                 draw second until second to last nodes in current_path
71:                 update vis_positions to contain positions of new nodes
72:                 draw edge into current_path (between first and second node)
73:                 draw edge out of current_path (between second to last and last node)
74:             end if
75:             append current_edge to done_edges
76:             remove i from further_edges
77:             break
78:         end if
79:     end for
80: end while
```

Both when generating GML data as well as when visualizing an automaton, we aim to identify one main path from the start node v_s to the end node v_e, and to then identify further paths lying off this main path. When generating GML data, these additional paths are added as info blocks, while we add them as paths of nodes on different horizontal lines when visualizing the automaton.
Having a closer look at line 5 of the algorithm, it can be seen that it constructs the main path by starting in the first node, which is assumed to be labelled with the hash tag symbol. In line 20 of the algorithm it can then be seen that to construct the main path 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, which is given if the graph is acyclic. 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 cycles, as it might otherwise be trapped in the cycle indefinitely, trying to find yet longer paths. An example for how this choice affects the visualized graph can be seen in figure 3.14. The graph nodes are then adjusted

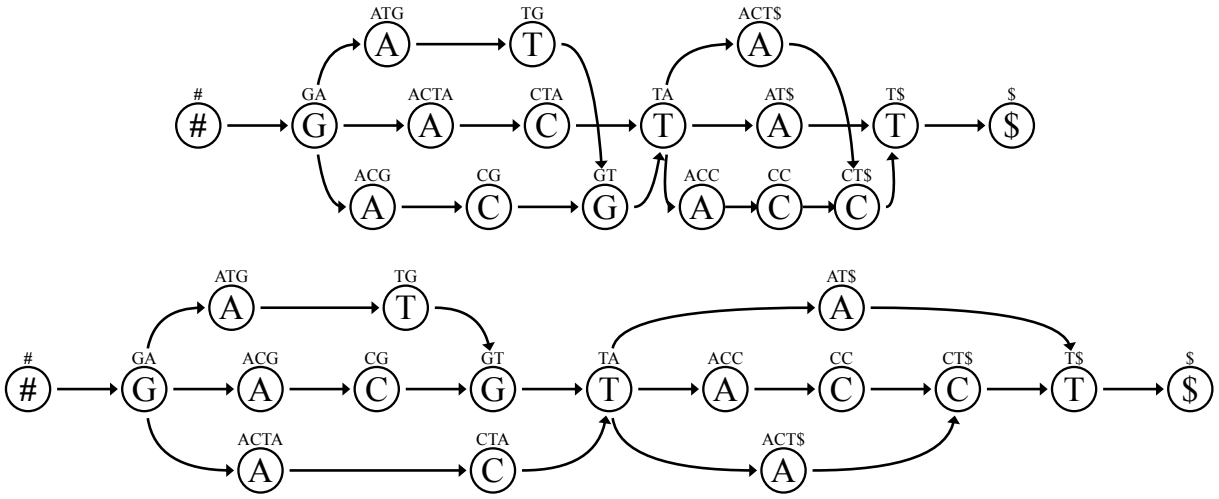

Figure 3.14: Two visualizations of the same graph. The visualization on the top uses the shortest possible path as main path, while the visualization on the bottom uses the longest possible path.

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 43 we iterate over all edges that have been added to further_edges, an array which keeps track of edges which we have not been drawn yet.
Line 46 ensures that we only proceed with an edge if it starts in a node which has already been drawn. However, the edge might lead to a node which has or has not yet been drawn. If the node it leads into has already been drawn, then we simply connect the start and end node of this edge in line 68.
If the node it leads into has not yet been drawn, then the iteration in line 51 is used to append nodes to current_path until we read a node which has been drawn, therefore giving us an entire path of nodes which all have not been drawn, starting in a drawn node and ending in a drawn node.
Algorithmically it would be much more straightforward to simply iterate over all nodes 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 edges that have not been worked on and nesting into this an iteration over nodes which have not been drawn 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. Direct Graph Merging

The aim of the Graph Merging Library is to provide advanced algorithms for the direct merging of flat XBW tables. However, it is helpful to be able to quickly check whether this merging of tables was successful. We therefore developed the function `mergeAutomata`, the general approach of which is shown in algorithm 3.5. It is used to merge two input graphs which are stored in the internal graph format outlined in section 3.4.2.
The function operates by first copying every node from graph_1 into the newly created graph, except for the end node with label "$". A list of all nodes which have outgoing edges leading to that node is hereby stored in the array out_of_1. This array is then iterated over, and from each of these nodes the edge leading to the end node is taken out. Finally, all nodes are iterated over and their pointers in the .p and .n arrays are updated to reflect the new positions of the nodes in the graph array which follow the now deleted end node. This concludes the deletion of the node with label "$" from the first graph.
Most of the nodes from the second input, graph_2, are then added to the resulting graph. Only one of the nodes from graph_2 is left out, which is the start node with label "#". As the start node is defined to always be in position 0 in the graph array, this can simply be done by starting the iteration at position 1. While adding the new nodes to the resulting graph, all their .p and .n arrays are updated using an offset. This is necessary as they get appended to the end of the new graph and therefore their positions are different than they were in graph_2. Equivalently to the array out_of_1, which kept track of the nodes with edges leading towards the end node in graph_1, we now also use an array into_2, which keeps track of the nodes with edges coming in from the start node in graph_2. Again, these edges are deleted to fully delete the start node from graph_2.
Having deleted both the end node of graph_1 and the start node of graph_2, we then attach the end of graph_1 to the start of graph_2, again using the out_of_1 and into_2 arrays which tell us which nodes lie at the end of graph_1 and at the start of graph_2, respectively. This concludes the function, as the merged graph has been fully formed.

Using this function to merge the graphs directly enables us to generate a flat XBW table based on the merged graph, which can then later on be compared against the results of the advanced XBW table merging algorithms to check whether they successfully merged the individual tables.

Algorithm 3.5 Merge two graphs in a simple manner. The input consists of graph_1 and graph_2, which are to be merged.

```
1: graph ← [ ]
2: offset ← (length of graph_1) − 2
3: into_2 ← graph_2[0].n
4:
5: for i from 0 to length of graph_1 do   ▷ Start to build graph as copy of graph_1
6:     if graph_1[i].c = '$' then
7:         del_i ← i
8:         out_of_1 ← graph_1[i].p
9:     else
10:        append graph_1[i] to graph
11:    end if
12: end for
13:
14: for i from 0 to length of out_of_1 do                    ▷ Take out link to $
15:    delete del_i from graph[out_of_1[i]].n
16: end for
17:
18: for i from 0 to length of graph do          ▷ Update indices after deleting $ node
19:    for j from 0 to length of graph[i].p do
20:        if graph[i].p[j] ≥ del_i then
21:            decrease graph[i].p[j]
22:        end if
23:    end for
24:    Repeat previous five lines with .n instead of .p
25: end for
26:
27: for i from 1 to length of graph_2 do                ▷ Add all nodes from graph_2
28:    for j from 0 to length of graph_2[i].p do
29:        graph_2[i].p ← graph_2[i].p + offset
30:    end for
31:    Repeat previous three lines with .n instead of .p
32:    append graph_2[i] to graph
33: end for
34:
35: for i from 0 to length of into_2 do                      ▷ Take out link to #
36:    into_2[i] ← into_2[i] + offset
37:    delete offset from graph[into_2[i]   ].p
38: end for
39:
40: for i from 0 to length of out_of_1 do                  ▷ Attach end of graph_1
41:    for j from 0 to length of into_2 do                    ▷ to start of graph_2
42:        append into_2[j] to graph[out_of_1[i].n
43:        append out_of_1[i] to graph[into_2[j].p
44:    end for
45: end for
```

3.8. Ensuring the Graph is Reverse Deterministic

Before being able to encode the graph as either XBW node table or as flat XBW table, it should be ensured that it actually is reverse deterministic.

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 Sirén et al. (2014) 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 $G = (V, E)$ not being reverse deterministic means that there is a node $u \in V$ which has at least two predecessors v and w which share a label. As v and w 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 v and w. As v is a predecessor of u, we can see that the prefix of v either continues with the prefix of u, or that it will not be able to continue after the first character as there is another successor of v which has a different label than u does.
The same holds true for w, such that both v and w have labels starting with the same character and are of length 1 unless they are that character concatenated with the prefix of u.
Now, in case of one of the prefixes just having length 1, we cannot sort v and w 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 u, as we can again not sort v and w 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 3.6.

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

Algorithm 3.6 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
```

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.

3.9. 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 3.15. If none of the nodes in the graph had more than one outgoing

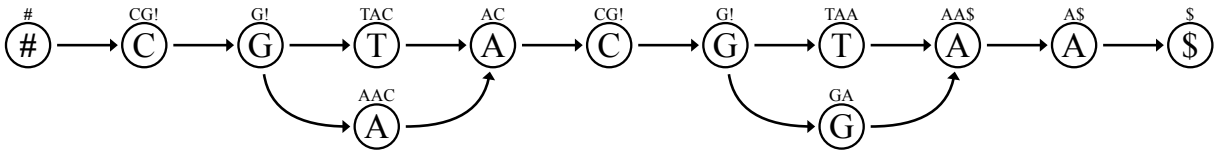

Figure 3.15: Reverse deterministic graph that is not prefix sorted. Among others, the two nodes with label "C" share the same prefix.

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 3.7. In line 18 of this algorithm, a node with more than one successor

Algorithm 3.7 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
```

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 3.16.

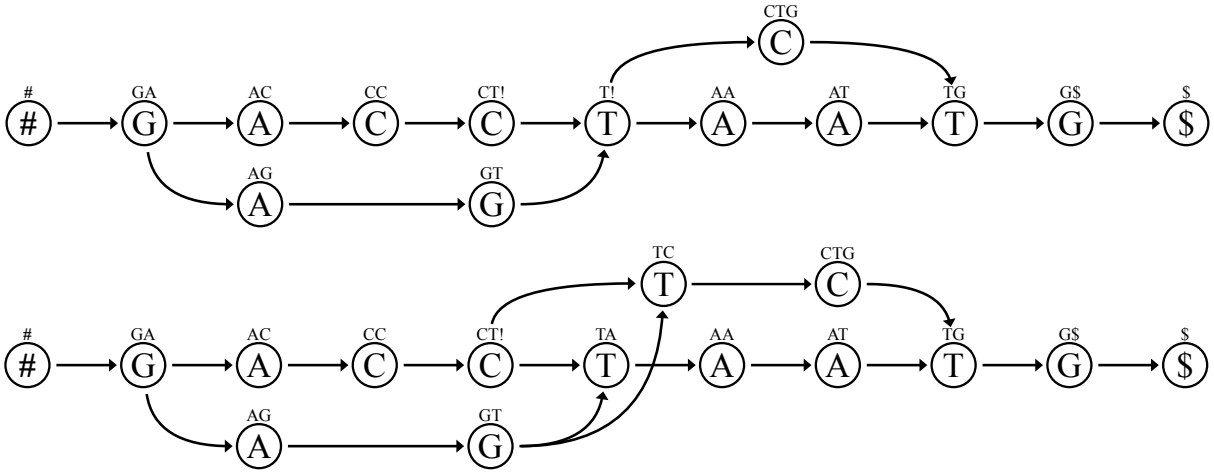

Figure 3.16: Node splitting example. On the top, the graph is visualized before the splitting of the central node with label "T" *and prefix* "T!". *On the bottom, the same graph can be seen after node splitting occurred. The node has been replaced with one node with prefix* "TC" *and one node with prefix* "TA". *Both nodes have all the incoming edges that the original node had.*

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.

3.10. Creating a Node Table

Ultimately, we want to generate a flat XBW table from the prefix sorted graph, and explore methods for merging these flat XBW tables. However, in this section we first consider node tables, which share several characteristics with flat XBW tables. Node tables are easier to understand and to use as they are more closely related to the underlying graph, which means that the visualized graph can be used to understand their behaviour. The definition of a node table can be found in section 2.7.1.

To create a node table when a prefix sorted graph such as the one in figure 3.17 is given, we can first create an alphabetically sorted list of all prefixes of the nodes in the graph. Such as list is shown in table 3.3.

Table 3.3: Alphabetically sorted prefix list for the graph in figure 3.17.

$	AC	AG	AT	C	GA	GT	T$	TAG	TAT	TGA	TGT	#	**Prefix**

Having created such a list, we can then iterate over the list of prefixes. For each node v_i corresponding to prefix i, we can add a cell for the value of the BWT and one for the

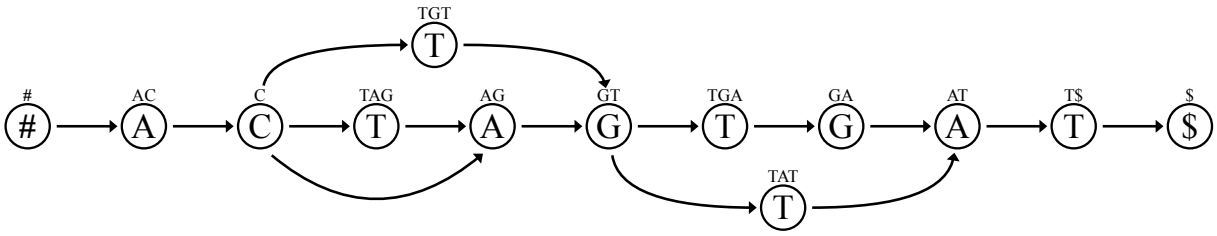

Figure 3.17: Prefix sorted graph, for which an XBW node table is supposed to be created.

value of the bit-vector M of v_i. We store the labels of all nodes immediately preceding v_i in the BWT cell, and a string starting with a one followed by some amount of zeroes such that its overall length is equal to the outdegree of v_i in the M cell.
This gives us a table containing the prefixes, BWT and M, such as table 3.4.

Table 3.4: Node table with BWT, prefixes and M for the graph in figure 3.17.

T	#	T|C	G|T	A	T	A|T	A	C	G	G	C	$	**BWT**
$	AC	AG	AT	C	GA	GT	T$	TAG	TAT	TGA	TGT	#	**Prefix**
1	1	1	1	100	1	10	1	1	1	1	1	1	***M***

We can finally also add a row for the bit-vector F, with each cell containing a string starting with a one followed by as many zeroes as are necessary such that the length of the string is equal to the amount of entries of the corresponding BWT cell. This concludes the generation of the XBW node table for the given graph. An example outcome can be seen in table 3.5.

Table 3.5: Complete node table for the graph in figure 3.17.

T	#	T|C	G|T	A	T	A|T	A	C	G	G	C	$	**BWT**
$	AC	AG	AT	C	GA	GT	T$	TAG	TAT	TGA	TGT	#	**Prefix**
1	1	1	1	100	1	10	1	1	1	1	1	1	***M***
1	1	10	10	1	1	10	1	1	1	1	1	1	***F***

3.11. Working on a Node Table

Assuming that we have created a node table encoding a genomic graph, we can perform various operations on it. This includes going from one node in the table to its successors or predecessors, as well as reading out the prefix of a given node to a certain length.

3.11.1. Navigating Through the Graph and Reading Out Labels

To navigate through a graph represented by an XBW node table, we should be able to find the start and end nodes within the graph as well as being able to go from a node to

a node connected to it through an edge.

To find the index i_s of the column representing the start node in a node table, we would usually search for all columns with a prefix starting with "#", as we know that the start node has a hash tag sign as its label.
As we assume there only to be one start node, there is only one such column in the node table. Its prefix therefore usually consists of just the letter "#", as there is no reason to build the prefix longer due to no other prefix starting with the same letter and an alphabetical ordering of the nodes by the prefixes being already guaranteed from just the first letter. However, a longer prefix would not be invalid, and therefore we would technically need to search for such a prefix starting with a hash tag sign.
Luckily, this entire search is unnecessary, as the single start node that is contained in the table can always be found in the same position. This position is the last, or right-most, column of the table. The reason why the column corresponding to the start node can always be found in this position is that the columns are alphabetically sorted by their prefixes, and the prefix of the start node starts with its label. The lexicographic value of this label is above the lexicographic value of all characters from the considered alphabet Σ.
Equivalently, there is only one end node v_e in the graph, and the column with index i_e corresponding to it can always be found in the very first, or left-most, column of the node table. Therefore, we always have $i_e = 0$. The reason is here that the lexicographic value of the label "$" is lower than that of all characters from the alphabet Σ.

The problem of advancing from a node v to the preceding or succeeding nodes can be thought about in the context of reading out the labels of these preceding and succeeding nodes. To that end, we can assume that we are interested in a certain column i of the node table, which corresponds to a node v_i in the graph represented by the table. Reading out the label $l(v_i)$ of that node itself can be achieved by reading out the prefix in column i of the table and taking the first character of this prefix.

To read out the label of a succeeding node, we actually have several possibilities. One way is to take the second character of the prefix of the current node, which corresponds to the first character of the prefix of each succeeding node, and therefore to their labels. This however only works if the prefix of v_i is given in the table with a length of more than one character, and if all succeeding nodes of v_i have the same label, as the prefix of v_i will otherwise have the error character "!" in the second position.
A similar approach can be used to read out the labels of the preceding nodes of v_i. All these labels are different, as the graph has been made reverse deterministic, and they are stored in the BWT row of the node table. So if we find $\mathrm{BWT}(i) = \mathrm{A|G}$, then we know that v_i has two preceding nodes with the labels A and G. This however only allows us to read out the labels of the immediately preceding nodes, while not enabling us to find the labels of nodes preceding the predecessors of v_i.

A more robust approach for reading out labels of preceding and succeeding nodes is to use navigation functions to traverse the graph from v_i to the succeeding or preceding node that we are interested in, and to then read out the label of that node as first letter of its

prefix.
To navigate the graph in the forward direction, that is, to find the immediate successors of a node v_i, we can use the `nextNodes` function. Algorithm 3.8 shows how this function works. The main idea behind this function is to add up all the lengths of the M values of

Algorithm 3.8 Given a column i in an XBW node table, find the indices of the columns corresponding to the succeeding nodes.

```
1: label ← prefix[i][0]
2: jump_over ← 0
3: for j from 0 to i do
4:     if label = prefix[j][0] then
5:         increase jump_over
6:     end if
7: end for
8: outdegree ← length of M[i]
9: nodes_found ← []
10: for k from 0 to length of BWT do
11:     if BWT[k] contains label then
12:         if jump_over > 0 then
13:             decrease jump_over
14:         else
15:             decrease outdegree
16:             append k to nodes_found
17:             if outdegree < 1 then
18:                 return nodes_found
19:             end if
20:         end if
21:     end if
22: end for
```

the columns in the node table to the left of i which have the same label as v_i does. The resulting integer is stored in the variable jump_over, as we intend to jump over that many nodes in the next step. We then iterate over the node table once more, this time reducing the amount of jump_over by one for each column which contains the label of v_i in its BWT cell. When jump_over reaches zero, we start adding the indices of the columns which we find to the output, until we have added as many indices as the outdegree of v_i. This approach works based on the fact that the left-most occurrence of a letter in the labels of the nodes corresponds to the left-most occurrence of this letter in the BWT, the second occurrence of a letter in the labels of the nodes corresponds to the second occurrence of this letter in the BWT, and so on.

A similar approach can be used to find the preceding nodes of v_i, which has been implemented in the `prevNodes` function.

3.11.2. Constructing Prefixes

If we are given the index of a column in the node table, corresponding to a node in the graph, then we can easily read out its prefix as the content of the prefix row in the given column. However, we might need to construct a longer prefix. Such a situation could arise if we were searching for a given text of length n within the graph, while the prefix we get from the table only has length m with $m < n$. If the prefix agrees with the text for all the characters that are given, then we want to append more characters to the prefix until its length reaches n and a clear decision can be made on whether the prefix agrees with the text or not.
To construct such a longer prefix given a node table with short prefixes, we can use the idea of *prefix doubling*. Hereby we do not append one character to the prefix at a time by iterating over the labels of succeeding nodes, but instead we append the entire prefixes of succeeding nodes. To actually find these succeeding nodes, we use the `nextNodes` function introduced in the previous section.

3.12. 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.

3.12.1. Basic Approach

The general node table merging approach that we use in the function `merge_BWTs_advanced` is to add the columns of both tables which are supposed to be merged to a single table, to sort the columns in this new table by their prefixes, and to then expand prefixes until they all are unambiguous.

More exactly, we first add a row with the caption "origin" to both input tables, with all the cells of this row in the first table containing the character "0" and all the cells of the origin row in the second table containing the character "1". We use these rows to later on keep track of where a node came from, such that we can perform operations on both input tables even while they are joined together within the emerging new table.
We then concatenate both tables to get one merged table containing nodes from both input tables. We also replace the label of the end node of the first table with $\$_0$ and the label of the start node of the second table with $\#_0$, to easier keep track of which start and end node is really contained in the final graph and which one is only used while merging and will be discarded later on.
We can then sort all columns alphabetically according to their prefixes. If this does not lead to ambiguities due to columns with different origins sharing the same prefixes, then

we are lucky. However, if this occurs, then we have to continue constructing the prefixes of these ambiguous nodes, until all the ambiguities have been resolved.
Finally, we can take out the $\$_0$ and $\#_0$ nodes which are no longer needed and drop the origin row to convert the table we were constructing into a fully merged XBW node table.

3.12.2. Aftersort Array

While working on the emerging XBW node table, we often have to construct longer prefixes for nodes with ambiguous prefixes. Do do so, we navigate through the graph and append labels of the nodes we encounter. However, this navigation through the graph is made more difficult by the fact that the ordering of the nodes of the first input table can be changed within the merged table. The reason for this is that their prefixes, which within the first input table ended at the end node v_e of the first graph, can now spill over into the second graph, thereby giving different relative orderings. This concept is illustrated in figure 3.18 and table 3.6.
We can see that in the input table at the top, the green node with prefix T$#C is sorted

Table 3.6: Node tables showing the purpose of the aftersort array. The upper two node tables are the inputs for the XBW node merging, while the bottom table is the outcome. The highlighted columns correspond to the highlighted nodes in figure 3.18.

T	G	A|T	G	#	A	C	$	**BWT**
$	AA	AT	C	G	T$#C	TA	#	**Prefix**
1	1	1	1	10	1	1	1	***M***
1	1	10	1	1	1	1	1	***F***

A	A|C	C	#	$	**BWT**
$	A$	AA	C	#	**Prefix**
1	1	1	10	1	***M***
1	10	1	1	1	***F***

A	A|C	C	G	A|T	T	G	#	C	A	$	**BWT**
$	A$	AA$	AAT	AT	CA	CT	G	TA	TC	#	**Prefix**
1	1	1	1	1	10	1	10	1	1	1	***M***
1	10	1	1	10	1	1	1	1	1	1	***F***

before the purple node with prefix TA, as the dollar sign is lexicographically smaller than all other considered characters. When considering just the first input table on its own, the prefix of the green node would actually be T$, as we only appended #C to that prefix is to show how the prefix will look like when a spillover into the second input graph occurs, in which labels from the second input graph are appended to prefixes in the first graph. However, in the merged table at the bottom, the green node with prefix TC is sorted after the purple node with prefix TA, as "C" is lexicographically bigger than "A".
Therefore, when we are working with the merged table, we cannot simply append the columns from the two input tables in the exact order in which they are within the input tables, but we need to change the order slightly to account for the fact that such spillovers from the first into the second input can happen. This itself is not a particularly challenging problem, as we can just account for this different sorting behaviour when initially constructing the merged table.

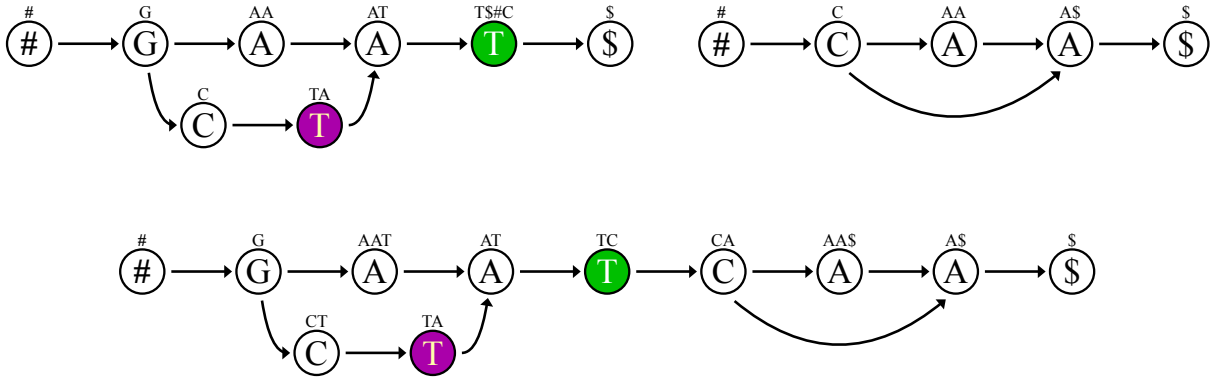

Figure 3.18: Graphs showing the purpose of the aftersort array. The upper two graphs correspond to the inputs for the XBW node merging, while the bottom graph corresponds to the outcome. The highlighted nodes correspond to the highlighted columns in table 3.6.

However, doing so results in a merged table which is internally inconsistent as long as we still see it as merged form of two individual tables, using an origin row to keep track of the partial table we are working in. To counteract this problem, we use a structure which we refer to as an *aftersort array*, which allows us to account for sorting problems after the start of merging the two tables. It provides a mapping from the locations within the first input table that we expect to the actual locations in the merged table, which can slightly differ from it. In particular, we use such an array with the name bwt_aftersort in the functions `nextNodes` and `prevNodes` to this end.
It is never necessary to apply an aftersort array when considering the second input, as no spillovers out of that table can occur when building prefixes, as we build prefixes by advancing to succeeding nodes, and no new node is succeeding the end node of the second input which actually becomes the end node of the merged table.

3.13. Creating a Flat Table

We now wish to create a flat XBW table based on a node table. For the general definition of such a flat table, see section 2.7.2.

The process of converting an XBW node table into a flat table is illustrated in tables 3.7 and 3.8, which respectively show the a node table and a flat table representing the same graph.

For the BWT, M and F rows, the process of achieving this conversion is exactly the same. We join 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

Table 3.7: XBW node table before conversion to flat table. Table 3.8 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 3.8: Flat XBW table after conversion from node table. Table 3.7 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***

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.

3.14. Working on a Flat Table

The next section is about the merging of several flat XBW tables. Before considering this rather challenging problem, it is helpful to first investigate how work within such flat tables can be performed in general. In particular, we here look at how the functions `LF` and Ψ can be used to find the predecessors and successors of nodes, respectively.

3.14.1. Graph Navigation

As with the navigation through XBW node tables described in section 3.11.1, the navigation through a graph represented by a flat XBW table relies on being able to find special nodes within the graph as well as being able to advance from one node to another one. In particular, we should to be able to find the start node and the end node in the flat table, and given a node v, we should be able to find the preceding and the succeeding nodes of v.

To find the cells corresponding to the start node v_s, we can again look at the end of the table, just as when finding the start node in a node table. However, in this case it is not enough to look at the last column, as some other cells could belong to the start node too. In particular, as the start node can have several outgoing edges, there could be several cells in the FC and M rows corresponding to the start node. Therefore, using FC-indexing or BWT-indexing complicates finding the start node slightly, while finding this node with absolute indexing is as simple as for node tables.
In absolute indexing, the index i_s of the start node is the highest index of all nodes. It can be found as $i_s = \text{rank}_1(M, l) = \text{rank}_1(F, l)$ where we define l as the index of the last column in the table, which is equal to one less than the amount of columns in the table.
Finding the cells corresponding to the end node v_e is even simpler, as its label "$" is lexicographically smaller than all other labels and it is therefore sorted to the very front. This means that the absolute index of the end node in a flat table is $i_e = 0$. As there can be no nodes preceding this one in the table, the index is also 0 when using FC-indexing or BWT-indexing.

To find the preceding nodes of a given node, we can use the function `LF`. It performs last-to-first mapping to map the last column in the matrix of sorted cyclic rotations, which is the BWT, to the first column, which is the FC row. How this function operates can be seen in algorithm 3.9. Its general approach is to take in a range $[sp, ep]$ of absolute indices of nodes $v_{sp}, ..., v_{ep}$ and a character c, and return a range of absolute indices corresponding to the nodes preceding the ones that were given as input which have character c as label. Both ranges can represent one or several nodes.
The `LF` function is more general than a function designed for just finding the preceding nodes of a given node. The ability of taking in more than one node through the input range combined with the ability to actively search for specific preceding nodes through the character c means that it can actually be used when searching for texts within the graph. However, it can of course also be used to just find the preceding nodes of a given node v_i, by calling it once for each BWT entry corresponding to v_i.
E.g. if two BWT entries correspond to v_i, then the function needs to be called twice, with the character c set to a different one of these BWT entries each time. The resulting ranges will describe one node each time, giving us a total of two preceding nodes for the input node.
We here made the select operations in the beginning and the rank operations in the end optional by introducing the parameters do_select and do_rank, so that the caller can set them to `false` after deciding to use different indexing methods if that is more convenient.

Algorithm 3.9 `LF` function for flat table navigation which takes in an absolutely indexed range $[sp, ep]$ and a character c. It gives out an absolutely indexed range corresponding to nodes with label c preceding the ones that were put in.

1: $sp \leftarrow \text{select}(1, F, sp)$ ▷ If do_select: absolute indexing to BWT-indexing
2: $ep \leftarrow \text{select}(1, F, ep + 1) - 1$
3:
4: $sp \leftarrow C[c] + \text{rank}(c, \text{BWT}, sp - 1) + 1$ ▷ Perform last-to-first mapping
5: $ep \leftarrow C[c] + \text{rank}(c, \text{BWT}, ep)$
6:
7: **if** ep $<$ sp **then** ▷ Return empty range if nothing is found
8: **return** [ ]
9: **end if**
10:
11: $sp \leftarrow \text{rank}(1, M, sp)$ ▷ If do_rank: FC-indexing to absolute indexing
12: $ep \leftarrow \text{rank}(1, M, ep)$
13:
14: **return** $[sp, ep]$

Similar to the `LF` function for preceding nodes, there is also a second special function we can use to find the succeeding nodes of a given node. This function is called Ψ and its approach is shown in algorithm 3.10. It works slightly different than `LF`, in that it does

Algorithm 3.10 Ψ function for flat table navigation which takes in an absolute index i and an edge number k. It gives out the absolute index of the kth node succeeding the node with index i.

1: $i \leftarrow \text{select}(1, M, i)$ ▷ If do_select: absolute indexing to FC-indexing
2:
3: $i \leftarrow i + k$
4: $c \leftarrow \text{FC}[i]$
5: $i \leftarrow \text{select}(c, \text{BWT}, i - C[c])$ ▷ Get the next node
6:
7: $i \leftarrow \text{rank}(1, F, i)$ ▷ If do_rank: BWT-indexing to absolute indexing
8:
9: **return** i

not work on a range of nodes, but instead focuses on a single node only. Similarly to `LF`, we need to call Ψ several times to find all nodes succeeding a given one. This functionality is provided through a second parameter which Ψ takes in addition to the index of the node we are examining. The second parameter determines which edge we take out of that node, and the index of the single node which we reach upon using that edge is returned. More formally, when i is the absolute index of a node v_i in the flat XBW table, and k is an integer, then $\Psi(i, k)$ returns the absolute index of the kth node succeeding the node v_i. So if the node v_i has two outgoing edges, then $\Psi(i, 0)$ returns the absolute index of the node reached when following the first of these edges, while $\Psi(i, 1)$ returns the absolute index of the node reached when taking the second outgoing edge.

The Ψ function described here differs slightly from the one proposed by Sirén et al. (2014). One of the differences is that we convert from absolute indexing to FC-indexing first, and then get the character c, as we directly take c from the FC row instead of using another structure which corresponds to the FC row without entries for which M is zero. The second difference is that we use $k = 0$ to indicate the first outgoing edge, while their function starts with $k = 1$. Finally, in our version we actually made the select operation in the beginning and the rank operation in the end optional by introducing the parameters do_select and do_rank, just as we did with the `LF` function. This in particular allows the caller to combine the parameters i and k into one value when calling with FC-indexing by setting do_select to `false`, therefore preventing unnecessary indexing method conversions.

3.14.2. Constructing Prefixes

Prefixes can be constructed using the Ψ function to advance from a node to the next ones, while keeping track of the labels of these nodes.
That is, to construct the prefix of a node v, we can read out the label $l(v)$ as first character of the prefix. We can then advance through all nodes u_j succeeding v by using the Ψ function once for each outgoing edge. If all these nodes u_j have the same label, so if $l(u_j) = l(u_k)$ for all j and k for which u_j and u_k are defined, then we can append this label to the prefix we are building. Otherwise, we have to append an error character such as "!" to indicate that this position in the prefix cannot be filled unambiguously and stop the prefix generation. We also have to stop building up the prefix when we add the special "$" character to it, as we would otherwise just continue building it from the start. We can continue onwards with this process until we have generated a prefix of the desired length by repeatedly using Ψ again to find all succeeding nodes of all current nodes u_j, and again adding the shared label of all of them if it is unambiguous.
This process is implemented in the function `_publishPrefix` of the XBW environment in GML. Its general approach is outlined in algorithm 3.11. As the input to the function is given with the FC-indexing method, a particular edge can be specified via which the first node is supposed to exited. This can be helpful for specific visualization purposes. However, usually this is not the intended behaviour, which is why we introduced the parameter use_all_out_edges which defaults to `true`. When it is left at its default value, all outgoing edges from the input node are considered, just as they should be to generate the real prefix of the node. Therefore, the parameter can usually be ignored. Only when it is explicitly set to `false` does the edge given by the input matter, as the other outgoing edges of the input node are then not visited while generating the prefix. The differences between the two possible choices for this parameter are shown in figure 3.19.

In addition to performing the aforementioned calculations to generate the prefix of a given node, this function can also return a recommended *split node* through the optional parameter give_the_split_node. The computations necessary for this have not been included in algorithm 3.11 to simplify the understanding of it, but they basically consist of an array of visited paths which keeps track of the potential split nodes, and an evaluation at the very end of the function which determines which one of these to return by choosing

Algorithm 3.11 Generate the prefix of a node with FC-index i in a flat XBW table up to a given length. The optional parameter use_all_out_edges can be set to `true` such that all outgoing edges of the specified node are used. Otherwise, only the specific edge addressed by i is followed out of the first considered node.

```
1: pref ← ''
2: i_arr ← [i]
3: for i from 0 to length do
4:     if (i > 0) or use_all_out_edges then
5:         len ← length of i_arr
6:         for j from 0 to len do
7:             if i > 0 then                                  ▷ Convert from absolute
8:                 i_arr[j] ← select(1, M, i_arr[j])          ▷ indexing to FC-indexing
9:             else
10:                decrease i_arr[j] while M[i_arr[j]] = 0
11:            end if
12:            for k from 1 until M[pref_cur_is[j] + k] ≠ 0 do
13:                append i_arr[j] + k to i_arr
14:            end for
15:        end for
16:    end if
17:    for j from 0 to length of i_arr do          ▷ Follow a specific outgoing edge
18:        i_arr[j] ← Ψ(i_arr[j], 0, false, false)     ▷ in FC-indexing and get the
19:    end for                                          ▷ answer in BWT-indexing
20:    char_to_add_now ← BWT[i_arr[0]]
21:    not_all_chars_are_the_same ← false
22:    for j from 0 to length of i_arr do
23:        if char_to_add_now ≠ BWT[i_arr[j]] then
24:            not_all_chars_are_the_same ← true
25:        end if
26:    end for
27:    if not_all_chars_are_the_same then
28:        pref ← pref + '!'                          ▷ Add error char to the prefix
29:        break
30:    else
31:        pref ← pref + char_to_add_now
32:        if (length of pref > 1) and (char_to_add_now = '$0') then     ▷ Spillover
33:            i ← (length of FC row in nextXBW) − 1   ▷ Find #0 node in next XBW
34:            return pref + nextXBW._publishPrefix(i)
35:        end if
36:    end if
37:    if (char_to_add_now = '$') or (char_to_add_now = '$0') then
38:        break
39:    end if
40:    for j from 0 to length of i_arr do          ▷ Convert from BWT-indexing
41:        i_arr[j] ← rank(1, F, i_arr[j])                ▷ to absolute indexing
42:    end for
43: end for
44: return pref
```

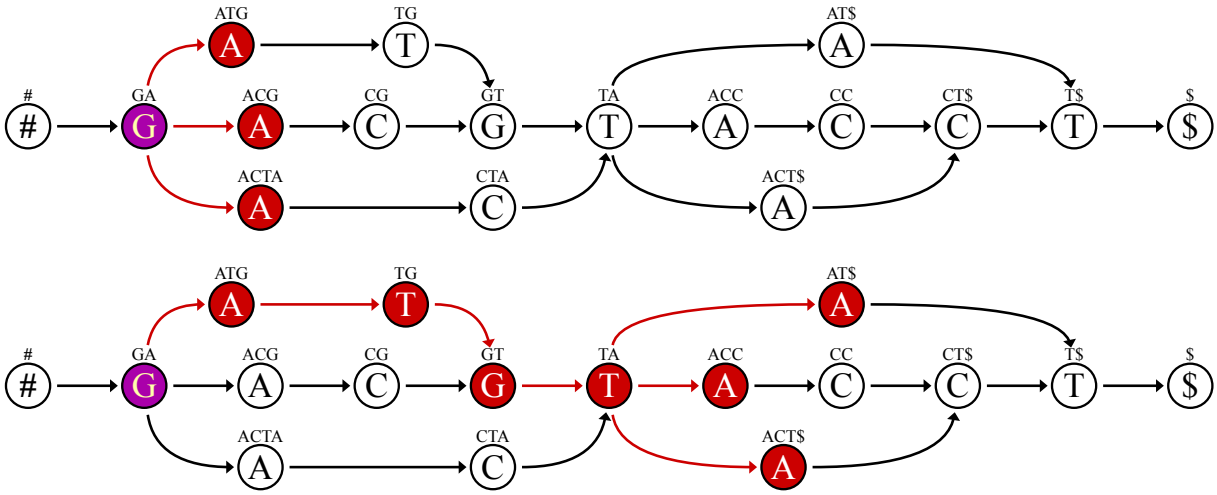

Figure 3.19: Prefix generation in a flat table with use_all_out_edges. *We here called* `_publishPrefix` *for the node with prefix* "GA" *highlighted in purple. At the top, we set* use_all_out_edges *to its default value* `true`*, and therefore the prefix* "GA!" *is returned. The prefix becomes ambiguous with the possibilities of constructing it as* "GAT" *or* "GAC". *At the bottom, we set* use_all_out_edges *to* `false` *and choose exactly one edge leading out of the purple node. This leads to the prefix* "GATGTA!" *being generated.*

a node actually on a path leading to a prefix ambiguity. We here define a split node as a node which can be split into several nodes, thereby ideally enabling us to continue building longer prefixes. An example for such a node can be seen in figure 3.20. It can

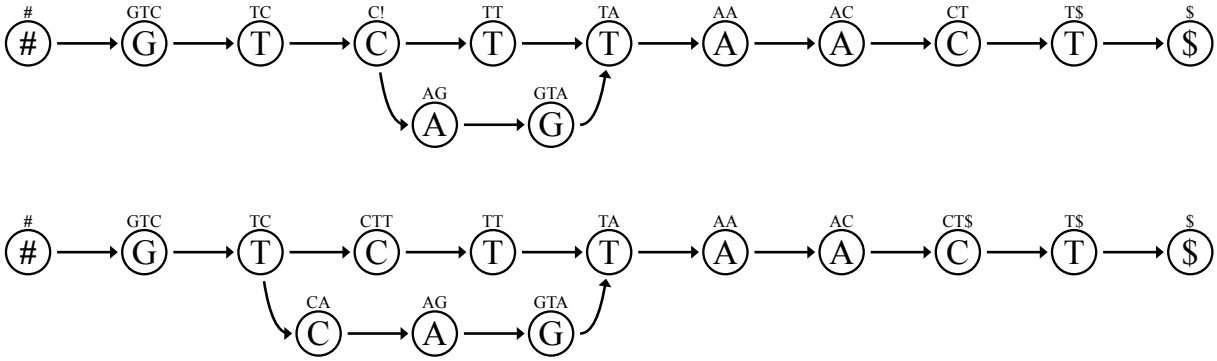

Figure 3.20: Graph with split node having prefix "C!" *before splitting (top) and the same graph after splitting the node (bottom).*

be seen that splitting the node with label "C!" in the figure did not change the genomic strings generated as paths through the whole graph from start node to end node, which is a general requirement we have for splitting nodes. The splitting helped in enabling us to build longer prefixes as one node with two outgoing edges to nodes with different labels was replaced by two nodes with only one outgoing edge each, so that each one of the new nodes can have a longer prefix.
For the general concept of splitting nodes to enable the building of longer prefixes, see section 3.9.

In figure 3.20, after generating the prefix of node "C!" the function `_publishPrefix` would return that node as a recommended split node. However, the node returned is not always the node for which the prefix is constructed; instead, we decided to return the first node with several outgoing edges encountered when going backwards through the tree of nodes that were inspected for generating the prefix, starting at the leaf nodes whose labels necessitated the adding of a "!" character instead of a letter. Such a tree of inspected nodes can be seen in figure 3.21. Here, we were trying to enlarge the prefix of a node

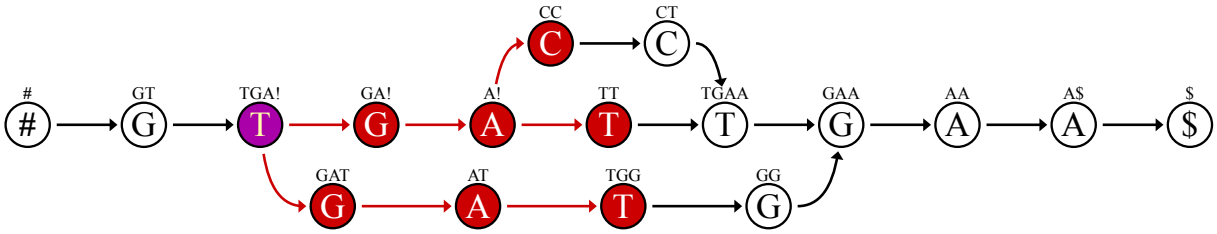

Figure 3.21: Graph with prefix generation tree based on node with prefix "TGA!"*, having leaves with labels* C, T *and another* T*. These leaves are also highlighted in red here, while the prefix tree visualization in GML usually leaves out the leaves of the prefix generation tree to only highlight the interior nodes which actually agree when building the prefix.*

with prefix "TGA!", and got the node with label "A!" as split node. It is not necessary to always act as the Graph Merging Library does in this case. It would be equally right to first split the node with prefix "TGA!" into two different nodes, and to split the node with label "A!" later on.

3.14.3. Finding Text Patterns

Text patters in a graph corresponding to a flat table can be found using the `LF` function with a big initial range which gets diminished through several applications, calling the function once for each letter in the text. This happens in the `find` function, the general approach of which is shown in algorithm 3.12. This function returns a range which corresponds to the nodes at which the given pattern P starts in the graph, given using the FC-indexing method. This method is used through the last parameter given to `LF`, which is $i > 0$. This parameter is do_rank, and it is `false` for the last iteration which we are looking at, so that `LF` in the last iteration does not perform the final rank operations to convert its output into absolute indices. The reason why we wish to use this indexing method here is that FC-indexing allows us to specify not just a node, but a precise outgoing edge of that node, so that the caller does not need to guess which edge to take out of the node. However, the caller is still expected to find the rest of the edges manually by traversing the graph from the node that was found onwards, as they are not directly given out.

Algorithm 3.12 Find pattern P in a flat table.

```
1: [sp, ep] ← [0, (length of BWT) − 1]
2: i ← length of P
3:
4: while i > 0 do
5:     i ← i − 1
6:     [sp, ep] ← LF([sp, ep], P[i], true, i > 0)
7:
8:     if ( length of [ sp, ep ] < 1 ) or ( ep < sp ) then
9:         return [ ]
10:    end if
11: end while
12:
13: return [sp, ep]
```

3.15. 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.

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. Therefore, merging the separate tables together as they are and working on the node splitting to expand the prefixes after the merge, as we did for merging node tables, is impractical. 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 three main steps, with the first being an iteration over all nodes 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 already within the input graphs before the actual merge. This is repeated until no further node splitting will be necessary when the actual merge occurs.
The iteration itself occurs as step-by-step comparison in which we advance through both input tables at the same time, always choosing to advance with the node that has the lexicographically smallest prefix of the considered nodes. This is done using the `checkIfSplitOneMore` function of the XBW environment in GML, which calculates one set of prefixes of nodes in both input tables and either advances to the next set of nodes, or populates the splitnodes array with nodes which it recommends to split to be able to construct unambiguous prefixes if this construction is not possible without splitting nodes.

To return such a split node, `checkIfSplitOneMore` internally uses the `_publishPrefix` function described in section 3.14.2, which can make a recommendation for a node that enables us to construct longer prefixes upon being split. If such a node is returned, then the function `splitOneMore` is used to actually perform the node splitting.
Just as when merging XBW node tables as described in section 3.12, we again use an aftersort array. The reason for using it is that the ordering of the nodes within the first input table is not necessarily the same as their ordering within the merged table, necessitating this adjustment to the order of comparison for the construction of prefixes and the splitting of nodes.

As the second and main part of the merging algorithm we then construct the merged table by again advancing through the input tables node for node and always adding the node with the lexicographically smallest prefix as next node to the emerging table. The way in which we step through the two input tables is exactly the same as in the first step, again using the aftersort array to allow us to append nodes to the final table in an ordering different from how they are ordered within the input.
We use the function `mergeOneMore` to advance through the nodes of both tables, and just like `checkIfSplitOneMore` it internally uses the `_publishPrefix` function to calculate one set of prefixes of nodes in both input tables. It then appends the node with the lexicographically lowest prefix to the emerging table and advances to the next set of nodes. It never has to return recommended splitting nodes instead of advancing to the next set of nodes, as the actions of the first step ensure that the two input tables can be merged in the second step without unambiguous prefixes occurring.

The third and final step in our implementation for merging flat XBW tables is to clean up the merged table by removing the end node of the first input graph and the start node of the second input graph. This is done in the function `finalizeMerge`. These nodes could have also not been added to the merged table in the first place, making it unnecessary to perform an extra step in the end. This however would have increased the complexity of the second step, which is why we decided to perform it as a separate step in the end.

3.16. Fusing Flat Tables

So far, we have investigated how several graphs can be merged together directly, how they can be merged when they are encoded as node tables, and finally how they can be merged when being encoded as flat tables. However, there is another interesting possibility for combining two graphs which we would like to look at. We decided to refer to this method as *fusing* several tables together, as it is inherently different from the actions we perform when merging tables.

3.16.1. Fusing Instead of Merging

As shown in section 3.15, merging flat XBW tables is rather complex. It requires a lot of computations and can lead to a lot of node splitting, which might be necessary to ensure unambiguously sortable prefixes along the entire merged graph. 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. How fusing differs from merging is illustrated in figure 3.22. This connection is not directly achieved within the flat table

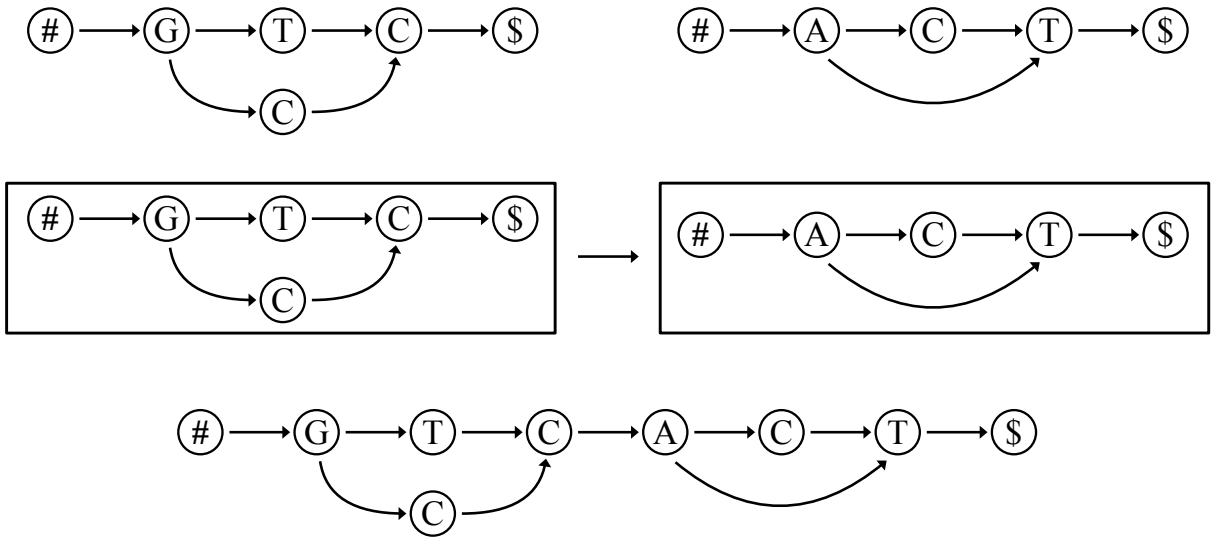

Figure 3.22: Merging and fusing graphs. The top row contains two input graphs. The middle row shows them fused together. The bottom row shows them merged together.

itself, but it is instead stored in a surrounding host data structure, and performed by the program working on it. The data structure itself just contains several completely regular flat XBW tables with one hash tag and one dollar sign node each, which in addition to the regular information can also store information about their immediate predecessors and successors.
That is, the core difference between merging and fusing is that when we merge two tables, we perform a lot of work once to generate one merged table. After the process is finished, it is impossible to decide whether the table has been constructed directly from some input graph, or has been the result of a merge operation of several graphs. On the other hand, when we are fusing two tables together, we only perform a relatively small amount work in the moment of fusing them, and we then perform a little bit of extra work every time we are operating on the fused table. The resulting data structure will always reveal that it actually consists of several individual tables, and the extra work needs to be performed every time that the table is operated on.
Implementing the fusing behaviour itself 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.
Also, we have to add an additional requirement to the tables which we consider. When we want to fuse tables together, then every fused start node needs to have exactly one outgoing edge, and every fused end node needs to have exactly one incoming edge. This requirement is necessary to ensure that the navigation functions `LF` and Ψ, which we will define for a fused table in section 3.16.4, can actually perform their work correctly. More elaborate future work could however erase this requirement by adding more end nodes and start nodes to each fused table representing the edges leading out of the start nodes and into the end nodes of the surrounding tables, respectively.
Apart from that, work can be performed in a fused table just like in a regular one.

3.16.2. Performing the Fusion of Two Tables

To actually perform the fusion two flat XBW tables, we create a host data structure which has a slot in which it can contain each table. We then make each table aware of its immediately succeeding table as well as its immediately preceding table. In the case of having two inputs, this means that the first input table needs to store the information that there is no preceding table, and that the second input table is its succeeding table. Similarly, the second input table needs to store the information that the first table is its preceding table, and that it has no succeeding table.
In GML, we use the function `initAsMergeHost` to initialize such a host data structure. It contains the array subXBWs, which in turn contains pointers to all tables added to the host data structure. The function itself just calls the general initialization function for any XBW environment, and then sets an internal integer called role to the value 3. Here, the value 1 stands for a regular XBW environment containing exactly one flat XBW table, while 2 stands for an XBW environment which is in the process of being merged from several tables and 3 stands for an environment which is the host data structure for several flat tables.
We then call the function `addSubXBW`, shown in algorithm 3.13, once for each table that we want to fuse together within that data structure. This function adds the table to the host data structure and connects the table to its predecessor. To do so, it first uses the functions `_setPrevXBW` and `_setNextXBW` to set both the preceding table and the succeeding table of the new table to `undefined`, meaning that the new table has no preceding or succeeding tables.
It then checks if the subXBWs array actually already contains entries, meaning that there already are tables stored in the data structure. If so, then the succeeding table of the last table in the structure is set to the new table, and the preceding table of the new table is set to that last table. If we are merging instead of fusing tables together, then the function also replaces the end node of the preceding table with "$\$_0$" and the start node of the new table with "$\#_0$". This however is not necessary when fusing tables, as the labels of nodes within the tables can stay exactly as they usually would, with spillover scenarios being completely handled from the outside when algorithms are working on the data structure.

Algorithm 3.13 Add a flat XBW table to a host data structure. The parameter newXBW is a pointer to the XBW environment containing the table which is to be added to the host data structure.

```
 1: newXBW._setPrevXBW(undefined)
 2: newXBW._setNextXBW(undefined)
 3:
 4: if length of subXBWs > 0 then
 5:     subXBWs[(length of subXBWs) − 1]._setNextXBW(newXBW)
 6:     newXBW._setPrevXBW(subXBWs[(length of subXBWs) − 1])
 7:
 8:     switch (kind of table combination)
 9:         case merging                                  ▷ Replace $ and # with $_0 and #_0
10:             subXBWs[(length of subXBWs) − 1]._replaceSpecialChar('$', '$_0')
11:             newXBW._replaceSpecialChar('#', '#_0')
12:             break
13:         end case
14:         case fusing              ▷ Throw error if several edges lead through fusion point
15:             if subXBWs[(length of subXBWs) − 1]._hasMoreThanOneEndNodeEdge()
16:                 throw invalid input error
17:             end if
18:             if nextXBW._hasMoreThanOneStartNodeEdge()
19:                 throw invalid input error
20:             end if
21:             break
22:         end case
23:     end switch
24: end if
25:
26: reset current positions within the new table to the start
27:
28: append newXBW to subXBWs
```

Otherwise, that is, if we are fusing instead of merging, the function checks whether there are several edges going through the new fusion location.
It therefore calls `_hasMoreThanOneEndNodeEdge` on the table which was the last one so far, and `_hasMoreThanOneStartNodeEdge` on the newly added table.
Finally, the new table is appended to the subXBWs array, which concludes the fusing of the table to the ones already in the data structure before.

Both of the functions used to check for multiple edges passing through the fusion point perform very simple checks which however require a fair amount of explanation.
The function `_hasMoreThanOneEndNodeEdge` is supposed to return `true` if the end node in the table on which it is called has more than one incoming edge, and `false` otherwise. It only contains the statement **return** $F[1] = 0$. This provides the required functionality as the end node has label \$, and its predecessors are found at the start of the BWT row. So if the F row starts with "11", then we know that the first element has only one incoming edge and we return `false`, as otherwise there would be one or more zeroes following the first "1". If the F row however starts with "10", then we know that the first element has at least two incoming edges and we return `true`.
The function `_hasMoreThanOneStartNodeEdge` on the other hand is supposed to return `true` if the start node has more than one outgoing edge, and `false` otherwise. It only contains the statement **return** $M[(\textbf{length of}\, M) - 1] = 0$. This provides the required functionality as the start node has label #, and it is found at the end of the FC row. So if the M row ends with "1", then we know that the last element has only one outgoing edge and we return `false`, as otherwise there would be one or more zeroes following the last "1". If the M row however ends with "0", then we know that the last element has at least two outgoing edges and we return `true`.

3.16.3. Fusing More Than Two Tables

When we want to merge more than two tables, we can simply do so by repeatedly merging pairs of two tables together. E.g. if we want to merge the tables representing graphs H_1, H_2 and H_3, then we can first merge H_1 and H_2 generating table H_{12}, and afterwards merge H_{12} and H_3 to generate table H_{123}. This means that although it might be interesting to look into merging algorithms which can handle more than two input tables at once, it is ultimately not entirely necessary to do so, as at least theoretically any number of tables can be merged if we are able to merge two tables.

When we intend to fuse several tables together, the situation is slightly different. As the process of fusing two tables together does not result in one table, but instead in a data structure internally consisting of two tables, we need to explicitly enable the fusing process to handle more than two input tables and fuse them together into one data structure. Furthermore each algorithm working on the fused data structure needs to be able to handle such a data structure containing an arbitrary amount of tables, as it would not help having a fused table consisting of more than two tables if there were no algorithms readily available to perform any work on that fused table.

Luckily, the changes needed to be made to the algorithm from section 3.16.2 to account for more than two input tables are rather small. In particular, the host data structure needs to be able to provide an arbitrary amount of slots into which to put these tables, such as an array which can hold as little as one element but has no practical boundary in how many elements it can contain. Also, the tables within the structure need to be made aware of the correct surrounding tables, each having exactly one preceding table and exactly one succeeding table, except for the first table which has no preceding table and the last table which has no succeeding table.
We here assume that a data structure representing a fused table always contains at least one table, as otherwise it would be pointless to try and perform any work on it anyway. The reason why we want to allow one table in the data structure instead of requiring at least two of them is that we would like algorithms designed to work on these data structures to also be able to handle single flat tables without having to create a whole special case for them in which they are used directly without surrounding data structure.

3.16.4. Navigating Through Fused Tables

Just as with the navigation through XBW node tables and regular flat XBW tables, we again wish to be able to find the start and end nodes of a fused flat XBW table. We also wish to able to find the preceding and succeeding nodes of a given node within a fused flat XBW table.

To be able to perform any of these navigation functions, we however first need to define an indexing method which can be used within the fused data structure. When working with a regular flat XBW table, we can use a single integer i to address a node via the three different methods of FC-indexing, BWT-indexing and absolute indexing. As all these methods have certain advantages and disadvantages, we would like to keep using them with fused flat tables.
In addition to the integer i, which gives us the location within a table, we however need to also keep track of which table the node is located in that we are addressing. One way is to simply use a second integer j to encode the table that we are interested in, such that each node in the fused graph is identified via a pair (i, j) of integers addressing the node in the table and the table within the structure.
However, a second way is to keep using a single integer, and to determine which table we are in by adding the lengths of all tables before the addressed one. So if we want to address a node with location i in the second table within a fused structure, we would actually address it as $|H_1| + i$, where $|H_1|$ stands for the amount of nodes in the first table.
We decided to use the first of these two methods, as it is not necessary to change all indices of succeeding tables if a node in a table gets added or deleted with this method. That is, we address each node in a fused table with a pair of integers, the first of which can be given via FC-indexing, BWT-indexing or absolute indexing, while the second one actually refers to the table we are picking.

Using this pairwise indexing method, we can find the start node of the graph as last node of the first table. We know that it is a node within the first table due to the fact that we start each path through the entire graph in the first table, and we know that it needs to be the last node within that table as we find the start node within any flat XBW table in the last position due to the lexicographical order of its label. The index of the start node v_s of the fused graph is therefore $(|H_{\text{last}}| - 1, 0)$, where H_{last} is the last table within the subXBWs array of the host data structure and $|H_{\text{last}}|$ is the amount of nodes it contains. It is just as simple to find the end node v_e of the fused graph. It is located in the last table as each path through the entire graph ends there, and within that table it is the first node due to the lexicographical properties of its label "$". The index of the end node v_e of the fused graph is therefore $(0, |\text{subXBWs}| - 1)$, where |subXBWs| is the amount of tables that the fused data structure contains.

Just as with regular flat XBW tables, we can also use the Ψ and `LF` functions to navigate through each table to find preceding and succeeding nodes. However, while the behaviour of these functions themselves within each table does not need to change at all, we do not call them directly in a fused graph.
Instead, we wrap them in special functions provided by the host XBW environment. In particular, we can call the function Ψ in the host environment with the same arguments as the regular Ψ function for a flat table, except for the first parameter in which we use a pair (i, j) to address the node we are interested in instead of a plain integer i. These same arguments include the optional parameters do_select and do_rank. The result of the function is also slightly different, again being a pair (i, j) to properly address a node within the fused graph instead of a plain integer i. The approach on which the Ψ function within the host environment is based is shown in algorithm 3.14.

Similarly we have the function `LF` which when called in the host environment accepts

Algorithm 3.14 Ψ function in a host environment for fused graphs. It takes in a pair (i, j) containing an absolute index and an integer representing a table within the fused graph. In addition to that, it takes in the integer k. This function gives out a pair (i, j), where i is the absolute index of the kth node succeeding the node with index i in the jth fused table.

```
1: i ← subXBWs[j].Ψ(i, k, do_select, do_rank)        ▷ Call regular Ψ function
2:
3: if (i = 0) and (j < (length of subXBWs) − 1) then    ▷ Found end node v_e
4:     j ← j + 1                                        ▷ Go to next table
5:     i ← (length of BWT in subXBWs[j]) − 1            ▷ Find start node v_s
6:     i ← subXBWs[j].Ψ(i, 0, false, do_rank)           ▷ Advance to node succeeding v_s
7: end if
8:
9: return [i, j]
```

a range of nodes and the index of a table as $([sp, ep], j)$ and gives out such a structure as well. It is shown in algorithm 3.15. As with the Ψ function, which had to be called on one table within the fused graph in particular, it can be seen that this function does not allow us to specify a range spanning more than one table. If calculations need to be

Algorithm 3.15 `LF` function in a host environment for fused graphs. It takes in a pair $([sp, ep], j)$ containing a absolutely indexed range and an integer representing a table within the fused graph. In addition to that, it takes in a character c. This function gives out a pair $([sp, ep], j)$, where $[sp, ep]$ is the absolutely indexed range corresponding to nodes with label c preceding the ones that were put in the jth fused table.

```
1: [sp, ep] ← subXBWs[j].LF([sp, ep], c, do_select, do_rank)    ▷ Call regular Ψ function
2:
3: if (c = '#') and ([sp, ep] is not empty) and (j > 0) then    ▷ Found start node v_s
4:     j ← j − 1                                                ▷ Go to previous table
5:     [sp, ep] ← [0, 0]                                        ▷ Find end node v_e
6:     c ← subXBWs[j].BWT[0]
7:     i ← subXBWs[j].LF([sp, ep], c, false, do_rank)           ▷ Advance to node preceding v_e
8: end if
9:
10: return [[sp, ep], j]
```

performed on several of the fused tables, such as for pattern searching across the entire graph, then the `LF` function needs to be called on each of the fused tables separately.

3.16.5. Prefix Construction

To construct the prefix a node within the fused table, we perform an algorithm similar to the one we performed for the prefix construction in regular flat XBW tables. In particular, we read out the label of the node itself as starting character of the prefix, and then append characters to the emerging prefix by using the Ψ function of the host environment to find the succeeding nodes and their labels. The only difference to the prefix construction in regular flat XBW tables is here that we have to take care of spillovers into succeeding tables, which can become especially confusing if we are following several paths which are not all within the same table.

3.16.6. Searching for Patterns

As with the other algorithms working on fused tables, also the search for a pattern in a fused graph is very similar to the algorithm for pattern searches on regular flat XBW tables. In particular, we perform a pattern search using the `LF` function of the host environment on each table contained in the host data structure, while keeping track of spillovers in which a pattern is broken across the borders of two tables.

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 than others for representing genomic graphs and in particular population graphs used as alignment references.
Explanations and examples for the different formats that we consider here are contained in section 3.1.

4.1.1. FASTG Format

In the FASTG format, genomic graphs can be encoded in a very flexible manner, making it possible to encode graphs which contain loops as well as graphs which consist of several unconnected parts. In addition to that, this format is also designed in such a way that any FASTG file containing a graph can easily be converted into a file just containing a linear sequence, such that backwards compatibility with software tools which cannot yet handle graph data is given.
However, the format also has several drawbacks, which we investigated into while implementing it.

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. While this does make it simpler to convert FASTG files into the linear FASTA format, it does add any value to a FASTG file as long as it is worked on in a pipeline of software tools which are all fully capable of working with graphs.
In the following example taken from a FASTG file, 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 unnecessarily complex. This can be seen by the fact that it allows for three different 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 handles 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, each program working on FASTG files has to implement different algorithms to perform the same work on the different layers.

Finally, complicating the implementation of the entirety of FASTG is also the fact that the format is open to amendments and still changing, 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, on which we based our bubble format, does not actually occur in the standard written in 2012 (Jaffe et al., 2012; Li, 2014b).

4.1.2. GFA Format

We also considered the GFA format in which genomic graphs can be encoded. This format is as flexible as the previously considered FASTG format, allowing us to encode graphs consisting of several parts which are not connected as well as loops.
An advantage of GFA over FASTG is that graph behaviour in GFA is encoded in completely the same fashion, no matter whether it is global or local. This simplifies writing software that can work with the format, as there is no need to implement algorithms in different forms to work on different layers.
Another advantage of this format is its flexible tag system, in which tags representing different kinds of additional information can be added to pretty much any line. As tags which are not known to the program opening a file can be safely ignored, this allows the format to grow into various specialized forms over time while still maintaining backwards compatibility. One use for such tags is the encoding of information about the alignment qualities associated with sequences and their links.

A potential drawback of GFA is that according to Heng Li, the original author of the format, GFA only aims to be an assembly format (Li, 2014d). This means that it is

aimed at representing a graph based on one particular read assembly, while it has not been designed to represent population graphs which can be used as graph references.
However, in practice this does not seem to be a significant drawback. During practical test implementations, the GFA format, despite not being aimed at representing reference graphs, could be used for this purpose without encountering notable difficulties.

4.1.3. Bubble Format

Surprisingly, implementing algorithms to open and save files in the seemingly simple bubble format was actually not easier than implementing more advanced formats like FASTG, GFA and GML. This format may look simple for short and uncomplicated graphs, but it can quickly become difficult to keep track of the depths at which some paths are nested when the graphs considered become more complicated.
Therefore, the key benefit of the bubble format, which is its general human-readability, is not conserved for bigger files, for which it simply represents a rather complicated format which is not capable of encoding many interesting kinds of graph behaviour.
Having experienced the complications arising from this seemingly simple format, we concluded that it would not be very helpful to use the bubble format within other formats to represent highly localized variations. Instead, approaches like in the GFA and GML format seem to be more useful, in which graph behaviour on any scale is encoded in the exact same way.

4.1.4. GML Format

While using the GML format within the Graph Merging Library, it proved to be a straightforward and flexible format. If we allow for loops then this format is as flexible as FASTG and GFA are, also being able to encode any kinds of genomic graphs. Therefore, the requirement of our input not containing any loops should be seen as a specific requirement for the Graph Merging Library, which attempts to convert such graphs into flat XBW tables in which loops can provide problems due to potential infinite loops during the prefix generation. It should however not be seen as a requirement for the GML format itself, which is perfectly capable of encoding graphs with loops.

A slight drawback that we noticed when using the GML format was that modifying graphs with it by hand is rather cumbersome. Especially deleting nodes on the main path is a quite elaborate process, in which the pointers of all paths leaving or entering the main path need to be updated. Other formats, such as GFA and FASTG, make this process a lot simpler, as entire sequences or data blocks are referenced and adding characters to such a sequence or deleting characters from it does not change how the sequence relates to the rest of the graph.
This problem however could be mitigated by implementing a special text editor for the GML format, which would be able to handle common functionalities such as removing a

node and inserting a node, while automatically updating the relevant pointers.

4.1.5. FFX Format

Lastly, we also considered the FFX format, in which flat XBW tables are stored directly. This format turned out to be very simple to implement when working with flat XBW tables already, as a file can be opened and used without performing much work. Equally, saving a flat XBW table in the FFX format is a quick and simple process.
However, manually modifying files in this format is not very intuitive, and the overall human-readability of this format is the worst of all formats considered here. This should not be seen as a huge drawback though, as we saw for many formats that the human-readability was only really useful for very small examples, and that large graphs are unintuitive in any format anyway.

We also noticed that none of the considered formats currently offer inbuilt compression techniques. Especially in the FFX format, changing this however would not be very difficult. As this format 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.

4.1.6. Format Recommendations

After having considered the five different formats mentioned so far, we can now make the recommendation to use the GFA format for genomic graphs if the ability of encoding additional information in tags is necessary, or to use the FFX format when working with flat XBW tables internally anyway, as this drastically reduces the amount of work associated with opening and saving files. The FASTG and GML format do not seem to provide drastic advantages over either of these formats, and the bubble format is not intended for real-world usage anyway.

4.2. Automated Tests

After having implemented the Graph Merging Library, we wanted to confirm how well it 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.

4.2.1. 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

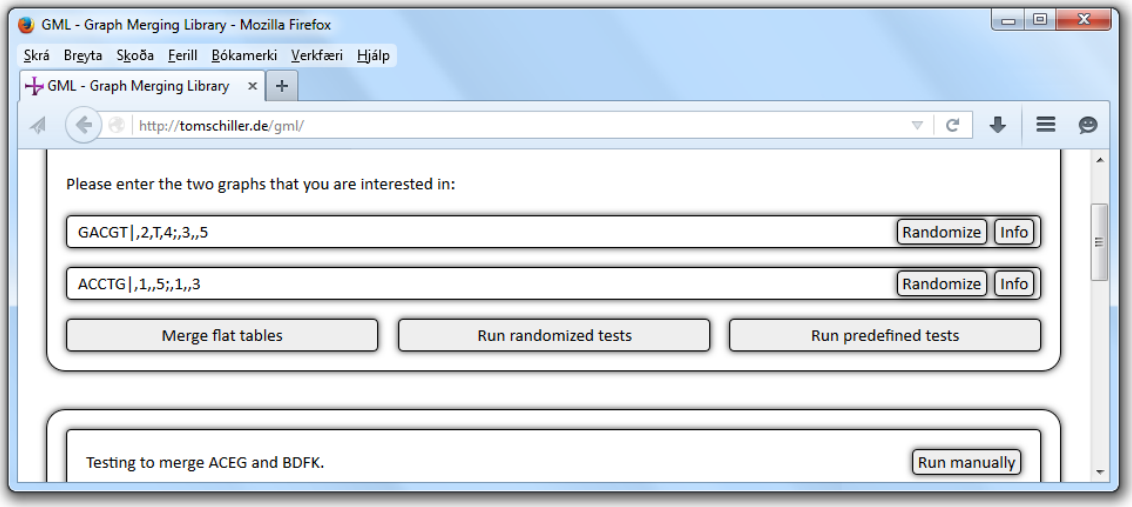

Figure 4.1: GML screenshot with row of three main action buttons. Clicking on the rightmost button starts the predefined tests, while clicking on the button on its left starts the randomized tests.

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.

A list of all predefined tests for the construction of XBW node tables can be found in

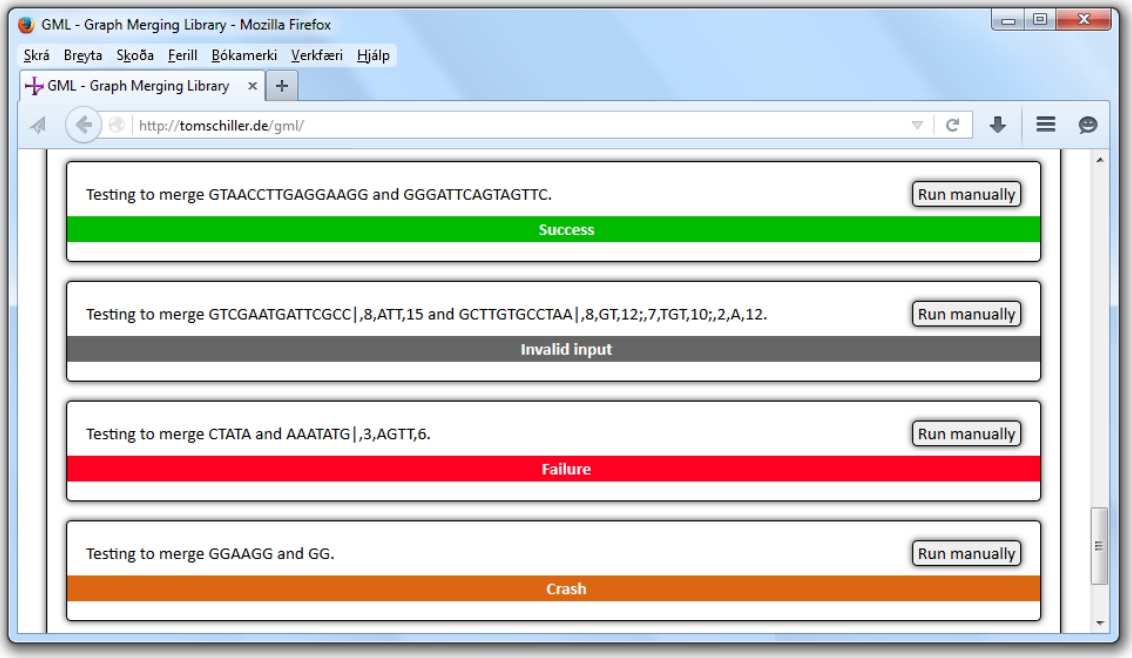

Figure 4.2: Different test results in the GUI of the Graph Merging Library.

section A.1 in the appendix. Section A.2 contains the predefined tests for the merging of XBW node tables and flat XBW tables. Each of these lists also contains a smaller optional section at its end, which contains graph inputs which are intentionally invalid. These are used to ensure that GML reports an input as invalid when incorrect graphs are entered.

4.2.2. 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 3.3. 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]

[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 u to node v only being allowed to be added if v could already be reached from u before the new path was added.

It is therefore necessary for the program to ensure that the input can actually be worked 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.

4.2.3. Test Analysis

For the Graph Merging Library to be able to report whether a test is a success or a failure, the more robust graph merging from section 3.7 is used to generate the result that we wish to see. Then, a more advanced algorithm is used for the direct merging of XBW node tables or flat XBW tables. The resulting flat tables are then compared. This approach is

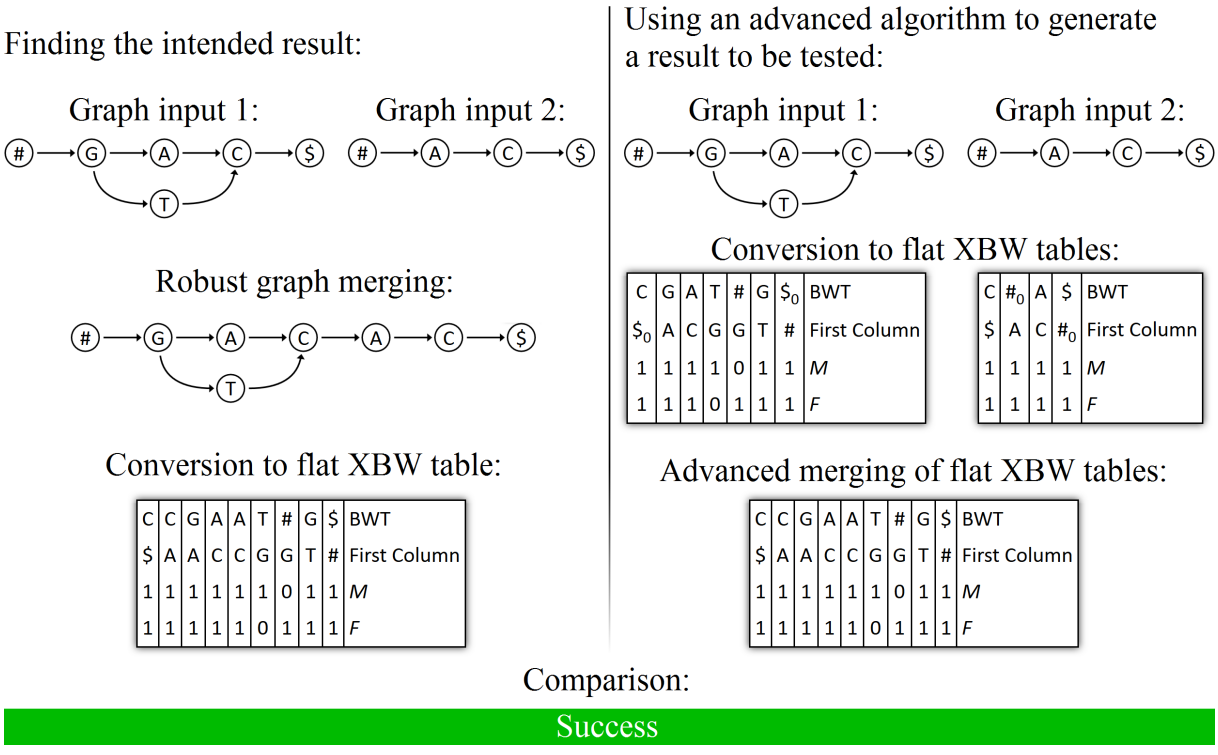

Figure 4.3: Test analysis by comparison of flat XBW table generated after robust merging of graphs and flat XBW table generated through advanced merging of several flat XBW tables.

illustrated in figure 4.3.
It is however entirely possible for the two tables to be different, but for the result to still be a success. That is, the flat XBW table generated through one method and the flat XBW table generated through another method can be different if the order in which BWT entries within a node are listed is different, as it does not make a difference in which order this listing is reported. Also, the tables can be different in case of nodes having been split by a method which are not entirely necessary to be split. This needs to be considered especially when prefix doubling is used for XBW node table merging, as the

prefix doubling by design propagates node splits which are not strictly necessary, in order to more quickly build prefixes.
To be able to compare the results of different methods anyway, and call a test a success even if not the exact same flat table has been generated, but a nevertheless equally acceptable table has been constructed, we convert the tables in question to the primitive automata described in section 3.4.2.
We further introduced the function `automataAreEquivalent`, which takes in two automata to give out `true` if both are equivalent, and `false` if they are not. We here define two automata to be equivalent if they generate the same language, even though they may differ in the amount and layout of nodes that they use to construct this language.
We can therefore check if both methods that are compared produce flat tables which correspond to automata generating the same languages, which means that graphs are generated which contain the exact same paths.

Even with the ability to compare the languages generated by automata, the automated tests still have to be analysed carefully instead of simply being taken at face value.
Consider a merge test with several randomized inputs showing a mismatch between the expected result and the actual result. Usually, this should 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. If the program worked correctly, this would be recognized automatically and the input would be flagged as invalid. However, if we would report this as a failure then there would be a bug somewhere in the program, so that bug could just as well be in the part of the source code which detects whether inputs are invalid or not.
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 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 both tests which are reported as failures as well as tests which are reported as invalid input should be manually investigated more closely to better understand the underlying cause of the categorization. Having said this, in several thousand testing rounds the categorizations have held up completely, and the reported outcome was indeed correct. A more detailed description of the tests which have been performed and analysed can be found in the next section.

4.2.4. Test Results

Running the over 100 predefined tests for the XBW creation, node table merging and flat table merging in Mozilla Firefox under Microsoft Windows 7 on a 2.40 GHz machine with 4 GB RAM resulted 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.

In addition to the predefined tests, we also executed 4,000 tests with randomized data, testing the merging results of the flat XBW table merging algorithm.
280 of these randomized tests were reported to contain invalid input, while the remaining 3720 tests were all reported as successes, either generating the exact expected flat XBW tables, or at least generating XBW tables which corresponded to graphs which contained exactly the correct paths.

It should of course be noted here that no amount of tests can ever truly prove the Graph Merging Library 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.

4.3. Merging XBW Tables

Using the Graph Merging Library, we have explored several ways to merge both XBW node tables and flat XBW tables. As could be expected, merging two node tables is a much simpler process than merging two flat tables, as a node table is less optimized for compression, 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.

4.4. Fusing Flat Tables Instead of Merging Them

When merging flat XBW tables, we perform node splitting to ensure that the merged graph is prefix sorted. However, each node that is split increases the overall amount of edges and nodes in the graph. Therefore, unless no node splitting is necessary at all, the merged graph has a size larger than the total size of the original graphs. An example for intense node splitting behaviour can be seen in figure 4.4. In this example, we have two input graphs with 23 edges in total. To save these together as fused graph, each graph is stored individually, such that 23 columns are stored altogether. However, if these were merged instead, we would obtain a graph with 50 edges, such that more than twice as many columns would need to be stored to save the merged graph. In addition to that, this is also an example in which the node splitting spills over across merging boundaries,

which means that the merging algorithms proposed earlier in this thesis would not be able to merge this graph anyway.

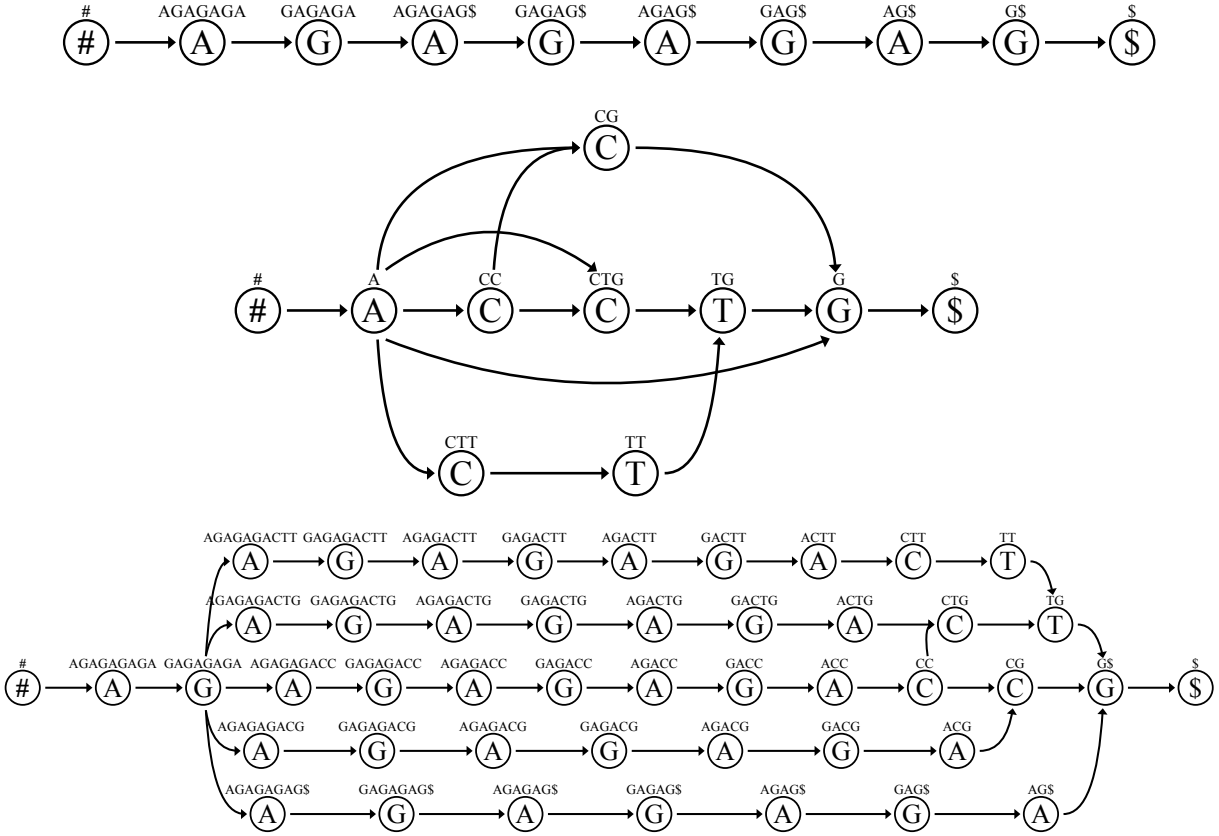

Figure 4.4: Node splitting resulting in a large merged graph. The top and middle graph are the input graphs, and the bottom shows the merged graph, which is much larger than each input individually. When fusing the input graphs together, no node splitting needs to occur, and their smaller sizes can be preserved in the resulting data structure.

The method of fusing flat XBW tables together as proposed in section 3.16 omits the need for an increased file size, as all input graphs are stored exactly as they are without having to perform additional node splitting. This not only makes it easier to store the data compared to storing a merged graph, but also means that working on a fused graph can be quicker as having less data means that fewer operations are necessary to construct distinct prefixes, to traverse all possible paths, and so on.

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.
Another disadvantage of fusing graphs instead of merging them is that we are only allowed to use end nodes with one incoming edge and start nodes with one outgoing edge at the

location of the fusion of any two graphs. This however is not a huge sacrifice, as fusing flat tables makes it unnecessary to split nodes, which we would possibly have to do when merging the tables. The splitting cannot occur across table boundaries, leaving merged graphs actually more restricted than fused ones.
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. Individually, handling each spillover scenario might be only a small amount of effort, but when large amounts of work are performed on the table, then these small amounts of work could add up to take significantly more time than work performed on a regular table would.
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.

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 performed 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 drastically spread the use of graph references. 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.
Internally encoding graphs as flat XBW tables and using formats such as FFX for working on these structures hereby seems like a promising method which could be used by more and more software tools over time. Ideally, the explanations of this approach given within this thesis will help steering more researchers in the direction of using this method to implement genomic graphs into their pipelines.

6. References

C. A. Albers, G. Lunter, D. G. MacArthur, G. McVean, W. H. Ouwehand, and R. Durbin. Dindel: Accurate indel calls from short-read data. *Genome Research*, 21:961–973, 2011. doi: 10.1101/gr.112326.110.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *Digital Equipment Corporation.* tech. rep. 124, 1994. URL `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html`. Accessed: 7th of April 2015.

K. Childs. Bioinformatics data formats, 2007. URL `http://rice.plantbiology.msu.edu/training/Childs_Data_Formats.pdf`. Accessed: 6th of January 2016.

P. Compeau and P. Pevzner. *Bioinformatics Algorithms.* Active Learning Publishers, La Jolla, CA, USA, 2014. ISBN 978-0-9903746-0-2.

G. M. Cooper. Switch from hg19/build37 to hg20/build38?, 2015. URL `http://www.reddit.com/r/genome/comments/3b3s3t/switch_from_hg19build37_to_hg20build38/`. Accessed: 27th of July 2015.

P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, 2000. ISBN: 0-7695-0850-2.

P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):Article 4, 2009. doi: 10.1145/1613676.1613680.

J. Holt and L. McMillan. Constructing Burrows–Wheeler transforms of large string collections via merging. *BCB '14, Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 464–471, 2014. doi: 10.1145/2649387.2649431.

L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013. doi: 10.1093/bioinformatics/btt215.

D. Jaffe, I. MacCallum, D. Rokhsar, and M. Schatz. The FASTG format specification, 2012. URL `http://fastg.sourceforge.net/FASTG_Spec_v1.00.pdf`. Accessed: 4th of April 2015.

J. Knight. Graphs, alignments, variants and annotations, pt. 1, 2014. URL `http://campuspress.yale.edu/knightlab/2014/07/28/gava-pt-1/`. Accessed: 6th of April 2015.

B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009. doi: 10.1186/gb-2009-10-3-r25.

C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002. doi: 10.1093/bioinformatics/18.3.452.

H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. 2013. arXiv:1303.3997 [q-bio.GN].

H. Li. Experimenting FM-index for multiple human samples, 2014a. URL `http://files.figshare.com/1614096/HengLi_20140717.pdf`. Accessed: 6th of April 2015.

H. Li. A proposal of the Grapical Fragment Assembly format, 2014b. URL `http://lh3.github.io/2014/07/19/a-proposal-of-the-grapical-fragment-assembly-format/`. Accessed: 4th of April 2015.

H. Li. First update on GFA, 2014c. URL `http://lh3.github.io/2014/07/23/first-update-on-gfa/`. Accessed: 4th of April 2015.

H. Li. On the graphical representation of sequences, 2014d. URL `http://lh3.github.io/2014/07/25/on-the-graphical-representation-of-sequences/`. Accessed: 5th of April 2015.

H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009. doi: 10.1093/bioinformatics/btp352.

V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010. doi: 10.1089/cmb.2009.0169.

E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(Suppl. 2):ii79–ii85, 2005. doi: 10.1093/bioinformatics/bti1114.

G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39 (1):Article 2, 2007. doi: 10.1145/1216370.1216372.

M. Nelson. Data compression with the Burrows–Wheeler transform. *Dr. Dobb's Journal*, September 1996.

P. Prins. Small tools manifesto for bioinformatics, Feb. 2014. URL `https://github.com/pjotrp/bioinformatics/blob/master/README.md`. Accessed: 27th of July 2015. doi: 10.5281/zenodo.11321.

S. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007. doi: 10.1145/1242471.1242472.

K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, and D. Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9):R98, 2009. doi: 10.1186/gb-2009-10-9-r98.

J. Sirén. Compressed suffix arrays for massive data. *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009 Proceedings*, pages 63–74, 2009. doi: 10.1007/978-3-642-03784-9_7.

J. Sirén, N. Välimäki, and V. Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014. doi: 10.1109/TCBB.2013.2297101.

F. W. Smith. Revised U.S.A. standard code for information interchange. *Western Union Technical Review*, November 1967.

A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform: the lively kernel experience. 2008. Technical Report, SMLI TR-2008-175, Sun Microsystems.

The GFA Format Specification Working Group. Graphical fragment assembly (gfa) format specification, 2015. URL `https://github.com/pmelsted/GFA-spec`. Accessed: 6th of January 2016.

The SAM/BAM Format Specification Working Group. Sequence alignment/map format specification, 2015. URL `http://samtools.github.io/hts-specs/SAMv1.pdf`. Accessed: 23rd of April 2015.

A. Appendix

A.1. Predefined Graph Construction Tests

The following tests are run to ensure that the Graph Merging Library does not throw errors while constructing flat XBW tables for input graphs:

ACEG
ACEG|,1,,3;,2,TB,4
GACGTACCTG|,2,T,4;,3,,5;,6,,10;,6,,8
GACGTACCTG|,2,,4
ACGTAGATTTC|,4,,7
TAATACGCGGGTC|,8,,9
GACGTACTG|,5,C,8;,2,G,5;,5,GT,9
GCTGGCGAGCAG|,3,T,6;,5,TC,12;,5,,7
GCTGGCGAGCAG|,3,T,5;,5,GCTC,12
GACGTACCTG|a,2,TAT,6;,a:0,C,a:2
GACCTAATG|,5,C,8;,2,G,5
ATTCACCACCAACCCGACATA|,4,CACG,9;,9,TCG,12
CCACGCGCCATGGC|,1,CTG,14
GCTGGCGAGCAG|,3,T,5;,5,,7
GCTGGCGAATGTGGCAAGAATGTGGCAAGAGCAG|,3,T,5;,5,,7;,4,CACG,9;,9,
TCG,12
GCCGT|,3,,5
ATCT|,2,A,3
ATCGAT|,2,,5;,4,CG,6
TGATGAG|,6,,7
GTTAATGTGGCAAGT
TCCCTGT
CTACCAGGTGCTGTTATTCCAC|,16,A,22
CTATGTTATTCCA|,8,A,13
TATA|,2,A,4
CCA|,1,A,2
TTCGCAAGA
CCTGCGTGTGGT|,3,,6
CCGAGCTTATCGCAA
ATGCGAGTTCGGGAC|,9,CCCG,14
GCAGGGAGCTCCAGCCGTTAG|,4,CC,13;,1,A,18;,17,,19

ATTGGAGAAGTCACGCTTGAC|,6,CTA,14
CACTC
TGCAAGTATGGCGCTT|,2,,8
AACCAT|,4,GGG,6
AACCAT|,4,GGG,6
AGAGCACGCCGGT
AGAGCACGCCGGT
CGGCAATAGACGCTGTCCAATGC|,15,,18
AGAGCAT
ACTGGCATGATTTATCCCTTGG|,11,,12
GGGCCCAGGGCGATCGACGTTC|,7,,19
CCCAGCC|,4,,6
CAGCC|,2,,4
CACT
CACT
TC|,1,C,2
ATCAACTTTCC|,6,CA,7
GCGGTGAAGAGAAA|,2,CGG,7
CCACGCTAAGTTATCGTGT
ATCCAATCGTAAT|,7,T,8
TAGCTTGGAC|,2,AG,9
GACAAACACAAACATCACCCTGT
CGTCGCC|,6,T,7
AAAGTAT
TACTTGTC
GCAGAATTCCGCAGGAAAGC
CGCGACAGTGGCCAATCT
GGTACATGGGAT
TCTGGTACGCTG|,1,,11
GGGAACAGATGTCTGTGATCC|,13,CAT,14
CGAGTTACGTGGCCGCCTCAT
TGTCTGTTACAGATTGC
CGAAGCCTA
CGAAAGAGTGTGTCTAGGC|,11,TC,19
TTTAGGTTAGTACCAC|,8,CTG,11
TACTTCCCCAGGACGGGACGCTA
AAAGGAGGT|,7,,8;,2,A,6
GGTGGCCGAGTGC|,10,CTGC,12
CGAATACCGTACTGAA|,12,AAAA,15;,14,GGTC,15
TTCCAATGGTGAGTCTC
AAGAT|,2,CTC,3;,1,C,4
AACAAATT|,4,TCCT,8
AAAAGAAGCTGAT|,4,,6
ACGTGCCCACCCG|,2,A,8
TAGGCGCGGT
TTTGCATCATATC

TTGTAG|,5,A,6
CTATATAGCGGC|,4,CTA,7
ATTAATACG|,2,G,5
CGGCCT|,1,A,6;,2,TT,4
CATCTTTC|,5,CA,7
AGGCAGCTATCGACCATCTTGCG|,22,,23;,2,TT,9;,9,C,10;,22,,23
AAGAGTCCAGAG|,7,,11;,5,TATA,9
GCCACGACACCCTCAAGCT|,17,CA,18
TTAAA|,4,GAT,5
TACCAGGGCTTTTTTACTGGCT|,8,,19;,20,GCT,21
TGAAGCGCATTCT|,9,,11;,3,,11
CGGATAGCACTCTA|,9,TT,11
TCCCTGGGC
CAAGGTATTGTTAA
CTCGGCTCT|,1,AAGG,2;,4,,6;,8,TG,9
AGGGAGCCTTAACATTTTCG|,13,T,19;,11,AGCG,15;,10,A,12
TCAGGGCGAGC|,6,G,9
GGAGCCAGGCTTGCCC|,7,,11
TGGCCCTCCCCCTACAT
AGGCCATTGATGAAAA|,7,AT,9;,5,CCT,12
GTATGCT|,6,TACC,7
ATTATCGATCA|,3,TCT,7;,10,CGCT,11;,8,TTCA,11
GTAACCTTGAGGAAGG
GTCGAATGATTCGCC|,8,ATT,15
CTATA
GGAAGG
GACGTACCTG|p1,1,AGCTTC,5;p2,1,AT,4;p3,5,A,8;p4,5,A,10;p9,p1:1,T,p2:1;p10,p2:1,
ACT,p4:0;,p9:0,G,p10:1;,p2:1,,p4:0

The following tests are run to ensure that GML correctly recognizes whether a dataset is an invalid input:

GACGTACCTG|,2,T,12;,3,,5;,6,,10;,6,,8
GACGTACCTG|,4,T,2;,3,,5;,6,,10;,6,,8
GACGTACCTG|,2,T,4;,a:3,,5;,6,,10;,6,,8
ATCTCAG|,1,GG,4;a,2,C,6;,a:1,G,7

A.2. Predefined Graph Merge and Fuse Tests

The following tests are run to ensure that the Graph Merging Library does not throw errors while merging XBW tables for input graphs. The same tests are also run for the fusing behaviour, just with a different selection of tests representing invalid input.

ACEG and BDFK

ACEG|,1,,3 and BDFK
ACEG|,1,,3;,2,TB,4 and BDFK|,2,,4
C and ACTG|,2,,4
C and ACATG
GACGT|,2,T,4;,3,,5 and ACCTG|,1,,5;,1,,3
C and ACTG|,2,T,4
GACGT|,2,C,4;,2,T,4;,3,,5 and ACCTG|,1,,5;,1,,3
GACGT|,2,T,4;,1,C,4;,3,,5 and ACCTG|,1,,5;,1,,3
TAT and C
TT|,1,A,2 and C
TCGTGCGAGG|,1,ACAA,10 and C
TAATACGCGGGTC|,8,,9 and ACCTG|,1,,5;,1,,3
AAGTTTCTTTCGTGCGAGGCCGT|,10,ACAA,19;,16,CGT,20 and ACCTG
ATAGTCAATTGACTGCCGACG and CGGGGTAAAAAAGCGCC
GCCG and GCGC
GCCG and BDFK
GG and CGG
ATCT|,2,A,3 and CC
ATCGAT|,2,,5;,4,CG,6 and C
TGATGAG|,6,,7 and GCCAGGTCAGCCTAGTCCCTG
GTTAATGTGGCAAGT and CTCGCGACGACTAAAGCTGGCC
TCCCTGT and CTCAGCAGAGGCCCAGGCAAA|,13,TA,15
CTACCAGGTGCTGTTATTCCAC|,16,A,22 and CATCGATTT|,3,CGAT,9
CTATGTTATTCCA|,8,A,13 and C
TATA|,2,A,4 and C
CCA|,1,A,2 and CA
TTCGCAAGA and TCCCCAAG
CCTGCGTGTGGT|,3,,6 and CTCTTA|,3,TCG,5
CCGAGCTTATCGCAA and GAGAGAGCAAC
ATGCGAGTTCGGGAC|,9,CCCG,14 and TCCAA
GCAGGGAGCTCCAGCCGTTAG|,4,CC,13;,1,A,18;,17,,19 and CGGAT
ATTGGAGAAGTCACGCTTGAC|,6,CTA,14 and GTTAAAACAC|,6,TC,10
CACTC and GGGCAGTACGTGG|,9,,11;,7,CGCG,8
TGCAAGTATGGCGCTT|,2,,8 and GTGTAATTATGAAC|,13,GATG,14
AACCAT|,4,GGG,6 and GTGCAGCTTTTGG
AACCAT|,4,GGG,6 and TTAAGGTTATGGACCCCCC|,4,GGAT,8;,6,C,12
AGAGCACGCCGGT and TTAAGGTTATGGACCCCCC|,4,GGAT,8;,6,C,12
AGAGCACGCCGGT and CGTAAATAGAG|,7,G,9
CGGCAATAGACGCTGTCCAATGC|,15,,18 and CGTAAATAGAG|,7,G,9
AGAGCAT and AGTGAACACAC
ACTGGCATGATTTATCCCTTGG|,11,,12 and TTTGTTTGACACGCTGC
|,16,,17;,11,AGGG,14
GGGCCCAGGGCGATCGACGTTC|,7,,19 and TTTGTTTGACACGCTGC
|,16,,17;,11,AGGG,14
CACT and GCGTACG|,5,,7;,1,C,5
CACT and GCGTACG|,4,,7;,1,C,5

TC|,1,C,2 and TC
ATCAACTTTCC|,6,CA,7 and TACCGAACGCCGGTGTGATAA|,6,,8
GCGGTGAAGAGAAA|,2,CGG,7 and GACTC
CCACGCTAAGTTATCGTGT and GCTTTATACGTAAT|,10,CC,13
ATCCAATCGTAAT|,7,T,8 and TCGAATTATCACGATAT|,13,AAA,17;,12,CG
,15;,7,,15;,7,GAG,16
TAGCTTGGAC|,2,AG,9 and GTGATGT|,5,,7
GACAAACACAAACATCACCCTGT and TGAAATAGT|,4,T,6
CGTCGCC|,6,T,7 and TCAATGTATAGTCGTGTCTGAAAT|,8,TGCA,23
AAAGTAT and AAAGCGAGCAAAAGATTGACGAA
TACTTGTC and CTCAACT|,6,,7
GCAGAATTCCGCAGGAAAGC and CATGGCGTCTAGGA|,10,GC,12
CGCGACAGTGGCCAATCT and TCGTCTGATGATCTTTGCTCTC|,19,,20
GGTACATGGGAT and ATGCGCTCGAAGCACGAAGG|,6,AT,15
TCTGGTACGCTG|,1,,11 and ATGCAAAAGTTTAAAGTTGTCCGT|,3,CGCT,18
GGGAACAGATGTCTGTGATCC|,13,CAT,14 and
TGAAAAAAACTCAGCGGGGAGT
CGAGTTACGTGGCCGCCTCAT and CCGTA|,3,TGT,5
TGTCTGTTACAGATTGC and AGACATCCATC|,4,,11;,10,TC,11
CGAAGCCTA and TTGTGAATATCG|,6,CTG,9
CGAAAGAGTGTGTCTAGGC|,11,TC,19 and AGAGCT
TTTAGGTTAGTACCAC|,8,CTG,11 and CATAGCTTAGCC|,11,,12
TACTTCCCCAGGACGGGACGCTA and GGTCAG
AAAGGAGGT|,7,,8;,2,A,6 and GAGTTATGACG
GGTGGCCGAGTGC|,10,CTGC,12 and CATTTGTGGCGCGCTTGATCTCTG
|,10,,20
CGAATACCGTACTGAA|,12,AAAA,15;,14,GGTC,15 and ACAGCGAGAGCAGACG
|,14,TC,15;,8,,16
TTCCAATGGTGAGTCTC and ATAAAG|,3,AG,5
AAGAT|,2,CTC,3;,1,C,4 and ACGCTCCTTTGTGG
AACAAATT|,4,TCCT,8 and GAGCGGAGAATCATACGGGC|,16,GAG,20;,1,CGC,17
AAAAGAAGCTGAT|,4,,6 and CTTGAA
ACGTGCCCACCCG|,2,A,8 and CCCCAACTGGGAATTC|,13,TCC,16;,4,TTT,14
TAGGCGCGGT and CTGGGACGG|,4,GTG,7;,2,C,6;,4,TGA,7;,6,C,9
TTTGCATCATATC and GAAACATTTATGACATTA|,4,,12
TTGTAG|,5,A,6 and AACGCGT|,3,,6
CTATATAGCGGC|,4,CTA,7 and ACGCATATTGGTTC
ATTAATACG|,2,G,5 and TTTTTATGGTCCGACTAA
CGGCCT|,1,A,6;,2,TT,4 and TTAATCT
CATCTTTC|,5,CA,7 and GTTGGTGGCCATCGG|,11,,15;,4,GG,14
AGGCAGCTATCGACCATCTTGCG|,22,,23;,2,TT,9;,9,C,10;,22,,23 and
AGCTGGGAAAC
AAGAGTCCAGAG|,7,,11;,5,TATA,9 and AGTGGCTGATTGGT
GCCACGACACCCTCAAGCT|,17,CA,18 and TCAATTGCCGGTA
TTAAA|,4,GAT,5 and GAACTCGGGGC|,3,TTGT,5

TACCAGGGCTTTTTTACTGGCT|,8,,19;,20,GCT,21 and
CCGGCGGATTGATAGTGC
TGAAGCGCATTCT|,9,,11;,3,,11 and GGATGCGAGCAGTACGTCTCCTTC
CGGATAGCACTCTA|,9,TT,11 and ATGTTGGGCCCTAA|,7,A,12;,8,T,11
TCCCTGGGC and ACGATTCCCTAACCACCCGT|,17,ATG,19;,7,TT,16
CAAGGTATTGTTAA and AGTCTAAGGCTTCGTGTTGGATC|,9,GTA,22
CTCGGCTCT|,1,AAGG,2;,4,,6;,8,TG,9 and AAAGTCTTCAATGT
AGGGAGCCTTAACATTTTCG|,13,T,19;,11,AGCG,15;,10,A,12 and
CATGTTGGTTGT
TCAGGGCGAGC|,6,G,9 and GACAACCCAGATTTTCCCGCATT|,10,,18;,15,TAA,17
GGAGCCAGGCTTGCCC|,7,,11 and ACAGGATAGGTCATTGAGTGGGG|,11,CAAC
,16
TGGCCCTCCCCCTACAT and TAGGCCGAAGTTTCTCTATCTTG|,13,CAAA
,18;,16,AAC,19
AGGCCATTGATGAAAA|,7,AT,9;,5,CCT,12 and GACTG|,2,,3
GTATGCT|,6,TACC,7 and GGACCTACGGCGGTTTAAAA|,8,A,14;,2,GCGA,12
ATTATCGATCA|,3,TCT,7;,10,CGCT,11;,8,TTCA,11 and
TGACAAGGTGCTCAGGTGAAA
GTAACCTTGAGGAAGG and GGGATTCAGTAGTTC
GGAAGG and GG
AATTCGGTAATACCGAAG|,15,AT,17 and TAAGATAAGGCAAA|,7,TA,11
CAGAGTATGCTCCGAG and CGATTACGCAT|,8,,10;,3,TA,9
GACGCCGACAAGAGACAAG|,2,CCA,11;,17,TC,18 and AGGTAGT
GTATAAATACAAGCTAAG|,17,CCA,18 and GATGA|,3,TCTC,4
CTAGA|,2,CC,3 and AAGATGAAAGGACCCCCA|,16,,18
GGGGAAAGCAAGATATCAGCCGT and CAAAAAGA|,4,,5;,3,C,5
ATGATG|,4,TC,6 and AAATGAGAT|,7,TGC,9;,5,AG,8
GCCAGA|,3,TAC,5 and CAACTCCAGGTTCCCTTATTG|,11,GTCA,13
GATATGGCGT|,3,A,9;,7,GTCA,10 and AAGACTTATACGC|,11,,13;,3,G,10
GCGGATATTTGGCGGCT|,6,A,14;,12,GTCA,15 and
AAGGAAGAAGACTTATACGCC|,18,,20;,10,G,17
TGGAAGCACACCC and TAGGGCGGAAGGG|,5,A,11;,5,TCCT,11;,8,GCCC,12
GTGCCCTAGACCTCCTTAT|,5,,10;,13,CCA,14;,7,GT,11;,3,G,8;,12,TGGG,18 and
GTGCA
TGTACTGGCTCCTATAT and TAACACATTAGTCAACCACA|,13,,16;,5,,12
CGACAG|,5,TTCC,6;,3,T,5 and GACCCACGCAAGAAACCGTGA|,10,TAAC,19
CGTTTGCATGAATTTGTTT and AACGTACCATTAAAATAGCGA|,6,,9;,4,TG,20
TACGATCGGA|,2,ATCG,6 and TAAGTTA|,2,CTG,3
GAGTATAAGCCA|,9,,10 and GGCGCGGTCAGTT|,12,GC,13;,2,GAT,6
AACCA|,4,CTTT,5;,4,CA,5;,4,ACG,5 and AGGTTATAGAT|,7,CAT,9;,2,,11
TCCGCTT|,2,TTTG,6;,4,ACT,6;,3,GGTG,6;,5,AACG,7;,2,ATC,5;,4,,5;,3,AA,6;,3,GGA
,7 and TCGACATTGCTGCAGGTT|,14,CT,18
CCCAGCC|,4,,6 and TT|,1,AGG,2
CAGCC|,2,,4 and TT|,1,AGG,2
TTCCGCGGCTCCATG|,14,CGTA,15 and ACCGTCCTCA|,8,CCGT,9;,8,C,10;,4,,10
ACTCTGAATGCTGTACACTCA|,5,,19;,13,C,14;,17,AGA,21 and GTTTGGGGA|,6,,9

ACCTG|,3,AAC,5;,1,C,2;,1,C,2 and ACGATGAAACGGATTGTA
TACACG|,4,AAT,6;,1,CCGA,5 and TGGTGTATTATTGCCCTTGGACGC|,20,GA
,22;,5,G,17;,8,AATA,10
TGTAGATGAGCATACCCCGA|,4,ACGA,20;,11,,19;,15,TT,19;,6,,11;,13,AA,20 and
AAATATCTTTGTTG
CTCATACGAGCGAA|,12,CG,13 and GTGCCCGGGTGCA|,1,TTTC,11
TTTTAATAGTCACCGATCCGG|,17,CGTT,21;,19,TTT,20 and TCGGCATT|,4,
ACGA,7;,6,GGA,8;,3,TGA,8
CCCAATGACAATG|,7,TAGA,10;,5,ACCC,10;,4,C,11;,4,,8;,8,C,11;,7,AGGT,9 and
GTGTC
GCTAGTTACAACT and GCAAGCGAA|,3,TCG,4;,5,T,8
CAGTTTACAACTGCAGCCGTTA|,7,,12 and TCGCCCGTAACTCAACGTTTAA
|,16,CTGG,17;,19,A,20;,19,GC,20
AACCCT|,1,ATCA,3;,2,C,4;,3,,5;,4,TCAT,6 and CCTTCGTACTAGCACAAACT|,5,
GGTT,11
TTGCTAACCAGC|,4,TC,5 and GGTGTTTGGGGCCCATACAAAT|,15,A,18;,1,GA
,11;,5,CCGC,14;,10,,19;,5,,13
GACGTACCTG|p1,1,AGCTTC,5;p2,1,AT,4;p3,5,A,8;p4,5,A,10;p9,p1:1,T,p2:1;p10,p2:1,
ACT,p4:0;,p9:0,G,p10:1;,p2:1,,p4:0 and ACCTG
CAA|,1,CG,3 and TGTC|,1,,4;,1,,3
GCCGTCAACAGAGCTCGTGCTA|,13,G,18;,3,GTA,10 and TCTCGT|,1,GG,6;,1,
GGCT,2
ACCACTCGGTT|,7,AA,8;,1,CG,9 and AAATCTTC|,2,,4
AACCGCGCCGTCTAAAGACCAA|,11,ACA,18 and TGGTCACCTCACTCCCTA
|,1,,10
AAAAGGAGGGATAC and GGATG|,2,A,5
GACGTACCTG|a,2,TAT,6;,a:0,C,a:2 and ACCTG|,1,,5;,1,,3

The following tests are run to ensure that GML correctly recognizes whether a dataset consisting of two graphs which are supposed to be merged contains an invalid input:

CTATA and AAATATG|,3,AGTT,6
GTCGAATGATTCGCC|,8,ATT,15 and GCTTGTGCCTAA|,8,GT,12;,7,TGT,10;,2,A
,12
TGGCC|,1,TA,2 and ACGCAGCTGCGAGAATA|,1,ACCC,5
GTAGCAAAGCGGTTTGGGATTG and GGTACCTATAGGGAA|,2,C,5
GAGTGGCCAAGTAGCCGGGTCACA|,23,GTC,24 and CGTACTG|,3,GAGC,7
GGTGGGATGGACTCTGTTGTG|,12,CACA,18 and CTGACTGTCAGGAGCTGA
|,3,,9
GGTACGCGTTATCGT and CTGTCTATCTCTAACCC|,12,,16;,2,T,6;,11,GA,12
GTAAC and TTTCACGCGT|,5,C,10;,5,CTC,6;,1,CG,3;,8,A,9
GTAATGACGTAGCCC and ATCATCGCA|,3,CT,8
GGGTTAGCTGTCTAAG|,6,C,9 and ATGATGATA|,5,ATAT,6;,1,TA,2
GTAGAGAATGGACTACTG and TGTGTGTTC|,2,ACG,4
ACGCAAGATGTATC|,3,GGAC,9;,10,G,11 and CCCCT|,2,CTT,5
TCGTCGTGAG|,5,G,6 and GAACCCGGAATCTTA|,2,T,15

ACTCGTGAAG and GGGCA|,1,CTCA,5
ATACGTAATCGGTGGCTCGA|,14,TCG,20 and AGATCGAGCTG|,4,,10;,8,,10;,2,
CTA,7
ATGTACACTTCTGAGTTGTCC and GCGTTGCG|,1,G,4
CGGAGACTATCCTATCC|,9,CAG,15;,11,,16 and
CAGTCATCGATTAAACCACTCTG|,1,GG,5
TGCTGCTTTGGAAAT|,1,ATTA,14 and GTCCGTAGCAA|,5,GG,7;,1,,7
GCTCTCATGCAGCCCAA and TTTCAGCCTAACAGTACACGGGT|,1,GA,19
CGCCC|,4,GAC,5;,4,,5;,3,ATG,4 and CAAGCTG|,1,GTTA,5;,5,GA,6
ACTCCTACTCCAACTA|,9,GTT,13 and ACTACC|,4,ATG,6
AACTAGGCGCATGTC|,3,CAG,8;,3,GTT,15 and CTTGGC|,3,AGA,6;,2,T,4;,2,CTAT
,4;,1,AT,2

A.3. 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 (Prins, 2014).
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 this thesis itself can be accessed at `http://tomschiller.de/reference_graphs`.