UiO : Department of Informatics
University of Oslo

Aligning reads against a graph based reference genome

Approximate searches in large and complex structures

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Aligning reads against a graph based reference genome

Esten Høyland Leonardsen

2nd May 2016

Abstract

As sequencing technologies improve, we are able to produce a larger amount of genetic data. One of the models used to organize and store this data are reference genomes, structures which collect such information to form a representative sample of the genome for a given species. To account for the variation which appears as the amount of data increases, new models for representing reference genomes are necessary. Graphs present the opportunity to have complex interrelationships between elements, a property which naturally solves the problem of variation. The newest human reference genome, GRCh38, already incorporate graph-like features through the introduction of alternate paths through variable regions. Methods created for interacting with the existing structures are traditionally centered around linear data representations, realized as a set of text string operations. To allow a complete transition, these methods must be adapted to fit the domain of graphs.

An important string operation in the context of genetic data is sequence alignment. In reference genomes, this is a technique which can be utilized for mapping new data against the reference. In this thesis, we present a new method for aligning text strings against graph based reference genomes. The method is built on the concept of context-based mapping; a technique proposed to standardize uniqueness in structures which do not have an innate coordinate system. We have made the method accessible through a tool which is available online.

We test the feasibility of our approach by doing performance comparisons with existing methods, examining both accuracy and efficiency. The results display several traits of the approach which outperform other proposed solutions. We argue that the method provides a viable solution to the most general version of the problem, which provides a basis for more specific biological applications.

Acknowledgements

I want to thank my supervisors Lex Nederbragt and Geir Kjetil Sandve for involving me in an interesting project and for providing guidance throughout the entire process. I also want to extend a special thanks to PhD candidate Ivar Grytten for proposing interesting views, good discussions and helpful tips and tricks along the way. I would like to commend the other master students in the Biomedical Informatics group on their contributions towards creating an exiciting learning environment. Lastly, I want to thank my friends and family for providing moral support and encouragement whenever needed over the last few months.

Contents

List of Figures

List of Tables

List of Theorems

Chapter 1

Introduction

With the initial sequencing of the human genome in 2001 [7], the blueprint of our species was made publicly available. Researches have utilized this data to understand the cause of genetic diseases and disorders and have made significant progress in the development of more accurate diagnoses and treatment. The current human reference genome, GRCh38 [15], has collected genetic information from several individuals to represent the human genome as a whole. Because DNA is prone to mutations, sequencing data which originates from a set of individuals is bound to contain variation. GRCh38 embed the concept of variation through allowing alternate paths through highly variable regions.

The transition from linear references to a structure which allows alternate paths is an effort to depict the variable nature of genetic information more accurately. As sequencing technology has progressed, the cost of sequencing has dramatically decreased. This has in turn lead to a larger number of individuals being sequenced, which results in the discovery of even more variation. Incorporating this variation into the reference genome leads to a higher level of precision and a better basis when it is utilized for mapping newly sequenced data. To achieve this goal, more general graphs have been proposed as a new standard for modeling reference genomes [5]. The data structure presented by graphs has the innate property of representing relationships which are more complex than what can be represented by linear models. The level of detail which can be achieved through a more complex representation can be of great value when mapping new data against the reference.

In order to understand genetic information, bioinformaticians have developed methods and tools for interacting with the data. To avoid a decrease in functionality when transitioning into a new model, these need to be adapted to continue providing a usable interface for researchers. This is a process which starts by formulating the problems in a language representable for the new domain and culminates in finding solutions to these problems.

1.1 Aims of the thesis

The master project in itself had a clear goal: Develop an algorithm for aligning text strings against graph based reference genomes. This also implies defining the problem in the context of graphs. The thesis will present both our formulation of the problem and the approach we propose as a solution. Interesting design choices taken throughout the process will be presented through the algorithm itself, the reasoning behind these choices given as formal arguments underway. Additionally, the thesis has two smaller goals:

- Validate the correctness of the approach
- Perform performance testing and comparisons to other tools on larger datasets

To succeed with the two smaller goals, we implemented the algorithm in the *GraphGenome* tool. This tool is available online, instructions on retrieving and using it can be found in Appendix C.

Throughout the development process, we were faced with several decisions regarding the specificity of the problem. In many of these situations, we chose to put an upper bound to the complexity, to end up with a simple, general, formally strict proof-of-concept, which should work as a foundation for expansions into more specialized applications. Every time we come across one of these simplifications we discuss the impact it has. In the later parts of the thesis, we reintroduce many of the concepts we have simplified away when discussing the feasibility of the approach in relation to more specific biological problems.

During the master project the article “Canonical, Stable, General Mapping using Context Schemes” [33] was published by Novak et al., discussing an approach to alignment which is similar to the one we present in this thesis. The similarities and differences between the two are granted a large part of the discussion section.

Chapter 2

Background

This chapter will build the foundation for understanding the theory necessary for the remainder of the thesis. We start out with a brief introduction to general biology. Because of the vastness of this field we only cover a small set of elements necessary for understanding the motivation behind the structure and approach presented later. A more thorough presentation can be found in the bibliography [21, Chapter 1 & 2][22]. We then step into the realm of technology and bioinformatics presenting techniques and concepts necessary in the description of our algorithm. We finish with a more specific presentation on the subject of graph based reference genomes, through discussing a set of articles, the solutions they provide and what we see as unresolved problems.

2.1 Genetics

Deoxyribonucleic acid (DNA) is a molecular structure in which living organisms store genetic information. The information is encoded by *nucleotides* bound together by a sugar-phosphate backbone into strands. The nucleotides are smaller molecules which vary based on the nitrogenous base they contain: *Adenine* (A), *Cytosine* (C), *Guanine* (G) or *Thymine* (T). Each of the nucleotides has a *complementary base*; A have T and C have G, with which it can bind to form a *base pair* (bp). Larger numbers of base pairs are typically prefixed as standard SI units[1]. Due to the chemical structure of the nucleotides, a DNA strand can be said to have a direction: Upstream towards the 5' end or downstream towards the 3' end. The DNA molecule is composed of two reverse complementary strands which are connected in a double helix structure. The two strands will have opposing directions, and every base in one of the strands will be joined in a base pair with its complement in the other. Because either of the strands can be easily deduced from the other, DNA is usually represented by only one of them. We can thus view DNA as a linear sequence of discrete units and represent it using text strings containing the four leading letters representing the nucleotides. The text strings representations often also contain the letter

[1] 1.000bp=1kb, 1.000.000bp=1Mb, 1.000.000.000bp=1Gb

N, referencing *aNy base*. The genetic sequence of an individual is called the *genotype*. Observable traits is called the *phenotype*.

2.1.1 The central dogma

The process of transforming the genetic information into large functional biomolecules is called *the central dogma* of molecular biology. The central dogma states that DNA is transcribed into *messenger RNA* (mRNA) which in turn is translated into proteins. mRNA is, like DNA, a sequence of nucleotides consisting of the three bases A, C and G and *Uracil* (U) instead of T. The mRNA can be divided into triplets of nucleotides called *codons*. The cell decodes the mRNA codons and creates strings of amino acids which are transformed into functional proteins. The relationship between codons and amino acids can be looked up in a table called *The standard genetic code* [22, Chapter 1, p. 6]. Only a portion of the nucleotides in DNA act as *coding regions* which make it through the transcription process and code for actual protein sequences. These are also called *exons*. The remaining *non-coding regions* of the genetic sequence are known as *introns*. In humans about 1.3% of the genome are coding regions [22, Chapter 4], the rest used to be referred to as *junk DNA*. We now know that the non-coding regions also holds important information [10].

2.1.2 Variation

Genetic information is prone to mutations, either as a result of environmental influence or as a consequence of imperfections during DNA transcription. The simplest mutations are *point mutations* which affect a single nucleotide base. Point mutations can either be *Single-nucleotide polymorphisms* (SNPs) where a single base is substituted for another or *insertions* or *deletions* (indels) where a single nucleotide is removed or inserted into the genetic sequence. Mutations can also occur over larger areas of the genome, where longer subsequences can be deleted, inserted, moved or reversed. A final type of mutations is *Copy number variations*, or *repeats*, where a longer sequence of DNA, typically at least 1 kb [12], is repeated a variable number of times.

As mutations happen randomly to individuals in a population, a diversity of genotypes emerges and creates variability within a *gene pool*. These different genotypes give rise to a variety of phenotypes. A subset of these phenotypes can ensure that an individual is better suited for survival and reproduction than others. Given enough time and scarcity in resources, the best suited individuals will survive and pass on their genes to the next generation. This is the process of *natural selection* which is the main driving force behind evolution. Another mechanism in play is *genetic drift* which affects gene frequencies in a gene pool through non-selective, random processes.

Because there are more possible combinations of nucleotide triplets than there are amino acids there exists some overlap between the codons and

the resulting amino acid. For instance, the DNA triplets “CGA”, “CGC”, “CGG”,“CGT”, “AGA” and “AGG”, all encode for the amino acid Arginine. In these cases, point mutations can occur without affecting the resulting protein. These mutations are called *synonymous*, the opposing case which alters the amino acid sequence is called *non-synonymous*.

2.2 Genetic data, sequencing and string algorithms

As genetic information is vital for determining the function of an individual, there is an obvious wish to understand this data. A large set of technologies and methods have been developed to collect and analyze the information. This section will present some important concepts for interacting with this data and techniques which play essential roles in the algorithm presented in the subsequent chapters of the thesis. We will also present the MHC region, the genetic region which provided the data we used in testing our approach.

2.2.1 Reference genomes

A *reference genome* is a data structure which contains genetic information for a population, typically for a given species. The reference genome has a set of continuous nucleotide sequences, called *contigs*, combined into larger *scaffolds* which again are combined to form the *genome* of the species. The first reference genomes collapsed samples from several individuals into a linear *consensus sequence* which was representable for the species as a whole. Later reference genomes have been built more flexibly to allow positions on the genome, called *loci*, to have several variants, termed *alternate loci*. A specific variant of a gene is called an *allele*. A *haplotype* is a set of alleles which tend to be inherited together. Reference genomes form what can be seen as an index for the genome of a species and can be used as a reference map when sequencing new genomes[2]. The structure of a reference genome can increase computational tractability compared to storing a set of individual genomes, by reducing double-storage of equivalent regions. The reference also provides a mosaic representing genetic variation, which can be useful when doing genetic analysis.

2.2.2 The human genome

The human genome consists of roughly 3Gb. The base pairs are spread over 46 chromosomes and are assumed to contain about 23 000 genes [22]. The human reference genome is developed and maintained by the *Genome Reference Consortium* [14]; the current version is called the GRCh38 [15]. GRCh38 contains 261 alternate loci, spread over 178 out of

[2]Covered in section 2.2.3

a total of 238 regions. An average human is estimated to deviate from the reference genome in 10.000-11.000 synonymous sites and 10.000-12.000 non-synonymous sites [9].

Major Histocompatibility Complex

The *Major Histocompatibility Complex* (MHC) is a genetic region spanning approximately 4.5-5Mb [11][33]. In humans, it is located on chromosome 6 and contains roughly 200 genes [1]. MHC is a region known to contain genes which affect the functionality of the immune system [44]. Even more so MHC is known to be a highly variable region, containing variants that are directly associated with disease [16]. The high variability creates difficulties when comparing DNA sequences to determine genetic causes of observed disorders, and when determining the origin of a sequence during *sequencing*.

2.2.3 Sequencing

In sequencing, a *sequencing machine* is used on a physical DNA fragment to find the underlying nucleotide sequence. The machines produce short *reads*, typically in the order of a hundred bp [38], which are combined into longer sequences through a process called *assembly*. When the sequenced individual belongs to a species which has a reference genome, reads are typically aligned against it to determine their position. The result is a set of variants, positions where the sequenced genome differs from the reference. These variants are often stored in a *Variant Call Format* (VCF) file, a file format constructed for this exact purpose. In cases where no reference exists, overlap techniques [36] or de Bruijn graphs[3] are often used in what is known as *de novo assembly* [22, Chapter 1, p. 19].

The different sequencing technologies have varying degrees of errors introduced in their reads [38]. The errors can take the form of both point mutations and larger structural variations. Reads produced by some high-throughput sequencing machines are typically prone to contain more errors towards the end of the read [39]. There exists efficient strategies for both estimating error rates [50] and correcting the reads [27]. These techniques can simplify the process of finding the origin of a read, which in a mapping assembly is done by an *alignment* algorithm.

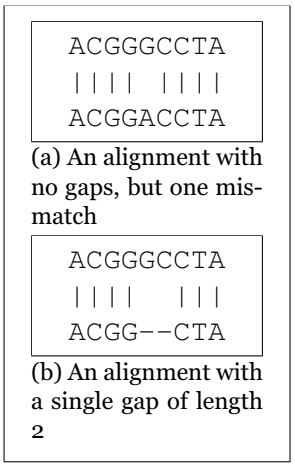
2.2.4 Alignment

Sequence alignment is the process of determining a correspondence between text strings, in this case representing DNA, by mapping the elements from one to the elements of the other according to a *substitution matrix* (Table 2.1) to provide a *mapping score*. Throughout this thesis, we will let the notation $mappingScore(c_1, c_2)$ denote the score for mapping two characters c_1, c_2 against each other. Alignment of DNA strings has several

[3]The concept is presented in section 2.3.1

important applications: As previously mentioned it is utilized as a tool for assembly as well as in genetic analysis and comparison.

The alignment procedure is never allowed to change the order of the elements in the two strings, but can introduce *gaps*. A gap occurs when a character in one of the strings is not mapped to a counterpart in the opposing string (Figure 2.1b). When a gap occurs, the resulting alignment is penalized according to the length of the gap by a *gap penalty*. We will similarly let the notation $gapPenalty(distance)$ denote the penalty achieved for a gap of length $distance$. Gap penalties come in different shapes, often according to the origin of the data involved. A *linear gap penalty* gives linear penalties related to the gap length. An *affine gap penalty* distinguishes between opening and continuing a gap. A *logarithmic gap penalty* lets the increase in penalty fade as the gap expands. We let which one of these is used, along with the choice of substitution matrix, be defined by a *scoring schema*. A scoring schema can thus also be seen as any structure which provides the $mappingScore$ and $gapPenalty$ functions. The scoring schemas can be based on simple match/mismatch scores, which corresponds to the mathematical *Edit distance problem*[4], or more complex scores (As depicted in Table 2.1). The complex models typically try to model the probabilities behind the physical processes responsible for change. The computational sequence alignment problem consists of finding the highest scoring alignment for any two strings.



(a) An alignment with no gaps, but one mismatch

(b) An alignment with a single gap of length 2

Figure 2.1

If more than two sequences are aligned, the result is a *Multiple sequence alignment* (MSA). This is typically done on sequences which are expected to share a common ancestor. The goal is to determine which traits in the individuals arose from the same origins and how the involved species have diverged genetically over time. A final variant of the alignment problem is one involving large databases of sequences, where the algorithms do not only need to find the best alignment between two sequences, but also determine which sequence should be chosen in order to maximize the result. Algorithms for both the preceding variants of the problem typically utilize heuristical methods to decrease the computational complexity.

There exist two distinct versions of the generalized problem: Finding *global alignments*, where two entire strings are aligned against each other, and finding *local alignments*, where a string is aligned against a substring of the other. The two are traditionally solved respectively by the Needleman-Wunsch and Smith-Waterman algorithms which both are based on *dynamic programming*.

[4]Covered in detail in the subsequent section

	A	C	G	T
A	91	−114	−31	−123
C	−114	100	−125	−31
G	−31	−125	100	−114
T	−123	−31	−114	91

Table 2.1: The HOXD70 substitution matrix [4]

2.2.5 Dynamic programming

Dynamic programming (DP) is a problem-solving technique where a problem instance is solved by breaking it into smaller subproblems and combining their results. DP is similar to recursion in that every instance is solved by a *recurrence relation* (An example can be seen in equation 2.1) which recurses on smaller and smaller problems until a *base case* is found. A base case represents the bottom of the recursion and is a value which can easily be computed without further recursive calls. The main difference between recursion and DP is that the latter usually stores its intermediate results to allow for fast lookups for reoccurring instances. DP is often used as an approach for optimization problems in order to minimize complexity while giving a guarantee for optimal results [2, Chapter 9].

A problem which is typically solved by dynamic programming is the previously mentioned edit distance problem (ED). ED is concerned with finding the minimal amount of substitutions, deletions, and insertions needed to transform one string into another. The algorithm utilizes a two-dimensional array to store the computed values. For two strings S and P, every index $[i, j]$ in the table represents the subproblem of finding the edit distance of the substrings $S[0:i]$ and $P[0:j]$.

		a	l	g	o	r	i	t	h	m
	0	1	2	3	4	5	6	7	8	9
l	1	1	1	2	3	4	5	6	7	8
o	2	2	2	2	2	3	4	5	6	7
g	3	3	3	2	3	4	5	6	7	8
a	4	3	4	3	3	4	5	6	7	8
r	5	4	4	4	4	3	4	5	6	7
i	6	5	5	5	5	4	3	4	5	6
t	7	6	6	6	6	5	4	3	4	5
h	8	7	7	7	7	6	5	4	3	4
m	9	8	8	8	8	7	6	5	4	3

Table 2.2: The two-dimensional array used for solving the edit distance problem for the strings S="algorithm" and P="logarithm"

In table 2.2, the base cases can be found in the first row and column. These are often dropped from the table itself due to the simple nature of their computations. The remainder of the table is filled out with the following recurrence relation:

$$D[i,j] = min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+score(S[i],P[j]) \end{cases} \tag{2.1}$$

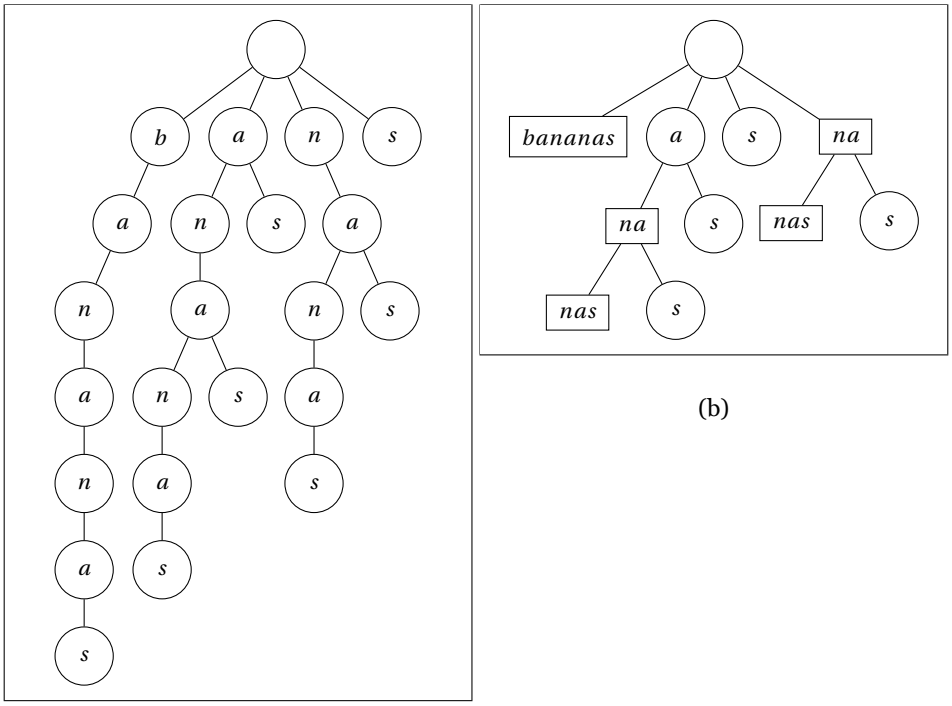
where $score(x,y)$ is an equality function returning 0 if the two elements are equal and 1 in all other cases. The score for the given instance of the problem can be found in the cell with the highest indexes in the bottom right corner.

There are two distinct ways of utilizing Dynamic Programming. A *bottom-up* approach starts at the smallest cases and computes everything until it reaches the actual given problem instance. This corresponds to starting in the top left corner of the edit distance array and computing the cells iteratively moving downwards to the right. A *top-down* procedure starts at the given problem instance and recursively computes every subproblem that is needed. This means starting in the bottom right corner of the two-dimensional array and recursing upwards to the left. For the edit distance problem the choice of approach bears no big significance as every cell has to be computed either way, but there are problems where using top-down can avoid some computations which are irrelevant to the final result. The latter can also be efficient for heuristical methods where an area of the search space can be overlooked.

2.2.6 Suffix trees

A *suffix trie* is a special tree constructed specifically for strings of text, containing vertices representing characters (Figure 2.2a). When creating a suffix trie for a given string, every suffix has a corresponding leaf vertex such that the vertices along the path from root to leaf contain the characters of that suffix. Consequently, every substring has a path starting in the root vertex. A *compressed suffix trie*, or *suffix tree*, is a suffix trie in which every linear path is compressed into a single vertice (Figure 2.2b)[5]. Both suffix tries and suffix trees can easily be extended to hold collections of strings [2, Chapter 20]. A naive implementation has a space complexity of $O(s)$, where s is the length of the string (or the total length of all strings if the tree is built from a collection), and a string of length m can be looked up in $O(km)$ time for an alphabet of size k [2, Section 20.6.1]. The tree can be constructed in linear time[47].

[5]The name "Suffix tree" is usually used as a general description of the concept. These are naming conventions from [52]



(a) (b)

Figure 2.2: The suffix trie (a) and suffix tree (b) of the string “bananas”

2.2.7 Approximate string searching

Given a dictionary and a query string, approximate string searching is the process of finding the elements in the collection which are most similar to the query string by some distance measure[6] [51]. This is a concept which is also referred to as *fuzzy searching*. When storage space is practically unlimited and time efficiency is the main priority, the problem is often solved through efficient storage structures [25, Chapter 3]. Suffix trees and *k-gram indexes* are examples of data structures used for this purpose. If the input data is user-generated search queries, these can be combined with statistical language models for enhanced performance. In applications where storage space is limited, dynamic programming methods can provide efficient solutions [42].

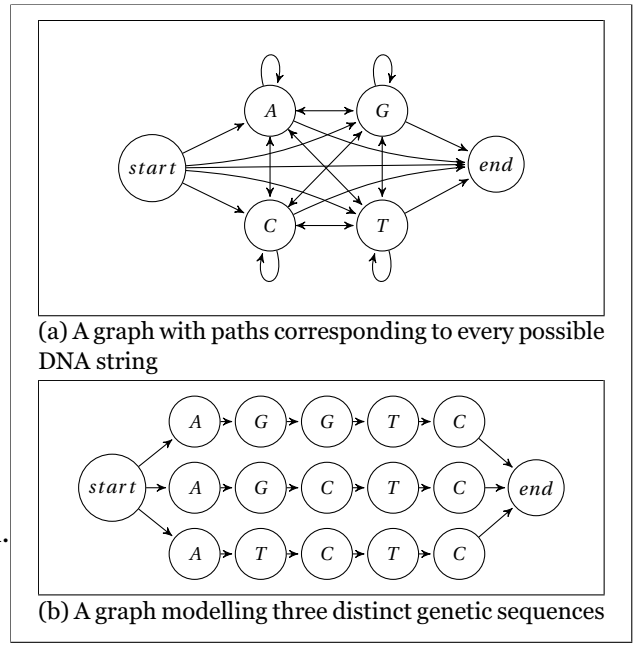
[6] ED is an example for such a measurement metric

2.3 Graph based genome representations

In the "Genetics" section we were introduced to the variable nature of genetic data. A linear model provided by text strings seems suboptimal for representing this variation due to its innate lack of flexibility. In this section, we will present graphs as an alternative model for genetic data. Graphs are far more expressive, and thus able to represent more complex relationships between the elements involved. Additionally, if we can rephrase biological questions in graph theoretical settings, we can benefit from the extensive mathematical field of graph theory when searching for solutions to arising problems. However, when changing the underlying structure, a major problem appears: To avoid a decline in functionality, the more complex model calls for more sophisticated variants of existing methods for interacting with the data. Graph-based approaches have been used for some time in the assembly process, and more recently in relation to reference genomes. We will present the work done on both these subjects alongside what we see as some of the remaining unsolved problems. The content of the section is presented in a way which should not require any prior knowledge of graph theory beyond elementary terms, but readers interested in a complete introduction is referred to the bibliography [52, Chapter 9][2, Chapter 11] [43, Chapter 0]. Complexity in regards to the graphs and their operations will throughout the thesis be discussed using *big-O* notation [52, Chapter 2][2, Section 3.1].

2.3.1 Model

Deciding upon the representation of the graph consists of defining the structure of the elements involved, namely the vertices and edges. As the graphs are built from genetic information, the basic building blocks, the nucleotides, should obviously be represented. If the input data is more complex than singular nucleotides, we must represent the relationships between them. Because the input data has variation, the structure needs to tolerate flexibility. There is, however, a risk of making the structures so



(a) A graph with paths corresponding to every possible DNA string

(b) A graph modelling three distinct genetic sequences

Figure 2.3: Two proposed graph models displaying flexibility (a) and rigidity (b)

flexible they present no consistency, and a flexibility/rigidness-tradeoff becomes apparent (Visualized in figure 2.3). How the structures are defined in detail should be determined through a requirement specification based on the operations which are desirable to perform on them.

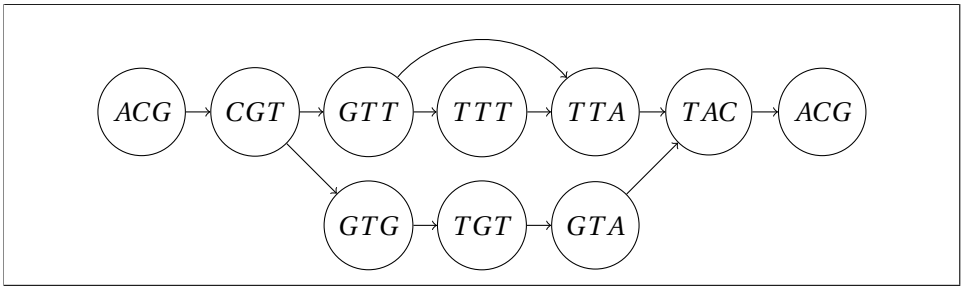
De Bruijn graphs

In the article "An Eulerian path approach to DNA fragment assembly" [36], Pevzner, Tang and Waterman proposes *de Bruijn* graphs as a solution to the problem of finding the correct origin of repeats during fragment assembly. A de Bruijn graph is a structure where vertices represent *k-mers* from an alphabet and edges represent relationships between the k-mers of two vertices (Figure 2.4a). Pevzner et al. let the vertices contain strings of length $l-1$ and connect vertices with an edge wherever there exists a read of length l containing both of the substrings. Formulating the problem in this fashion turns it into an *Eulerian path* problem, solvable in polynomial time, rather than the traditional "overlap-layout-consensus" method which is equivalent to the NP-complete problem of finding a *Hamiltonian path* [2, Section 11.1]. A great benefit with de Bruijn graphs is that there is no disambiguity: Any legal k-mer has at no point more than one vertice representing it.

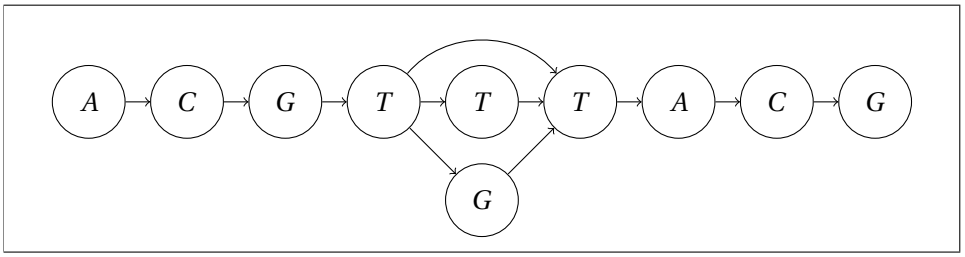
A more detailed type of de Bruijn graphs is the colored variant where the origins of edges and vertices are visualized as colors. The entire sequence originating from a single individual sample can be seen by following a path with a given color. Similarities between samples can be seen as multicolored stretches; variation takes the form of bubbles. Colored de Bruijn graphs can be used for de novo assembly as a more powerful method for detecting variation than traditional assembly techniques [17].

Sequence graphs

The relationship between a de Bruijn graph and the sequences it represents is not immediately apparent. A more intuitively pleasing representation is a graph where every vertice contains exactly one nucleotide (Figure 2.4b), a concept called *partially ordered graphs* by Lee et al. [20] and *sequence graphs* by Paten et al. [34]. In this representation, the underlying connection between the characters of the text string and the vertices of the graph is more apparent. The representation does, however, have a major disadvantage when compared to de Bruijn graphs: The concept of uniqueness. A vertice can no longer be identified solely by the data it contains. To solve this problem the vertices can be given ids, for instance UUIDs as proposed by Paten et al. Even though these IDs can be used to identify a vertex they contain no information regarding the relationships between elements. The difficulties presented by this problem will be the basis for the subsequent section on *mapping*.



(a) A de Bruijn graph with $k = 3$



(b) A Sequence graph

Figure 2.4: A de Bruijn graph and a sequence graph representation of the strings "ACGTTTACG", "ACGTGTACG" and "ACGTTACG"

2.3.2 Mapping

One of the operations which are necessary to perform, and therefore should be part of a requirement specification for the model, is mapping. We will in this thesis define mapping and alignment as two separate concepts, although the two terms are often used isomorphically. We let mapping be the process of finding relationships between single characters of a string and single elements of a reference genome. We define alignment as the process concerned with finding relationships between consecutive elements of an input string and substructures in the reference genome. In the context of linear strings, mapping is easy: Every string has the same underlying coordinate system, represented by the positions of the characters. This means two elements from two separate sequences are either in the same position or they are not. When they are not, the difference in position can be derived from the difference between the indexes, a measure of the distance from the beginning of the string to the position.

If we assume the indexation system proposed in the last section, the indexes of a graph have only one property: Uniqueness. They do not hold the intrinsic value of describing relations between vertices. A rigid mapping system using fixed coordinates would face problems when dealing with a fluent graph that can merge in new information as the internal relationships are bound to change. In de Bruijn graphs the problem is solved by moving the mappable quality away from positions and into the data: For any possible k-mer there either is a corresponding vertice or there is not.

In sequence graphs, where singular nucleotides are the most basic building block, there exists an equal number of identically scoring positions for every base as there are vertices containing that base in the graph.

Paten et al. [34] introduce the concept of *context-based mapping* as a solution to the mapping problem when the reference is modeled as a sequence graph. Context-based mapping is an approach where a vertex is identified by the surrounding environment in the graph. More technically a vertex has a set of *contexts* which are tuples (L, B, R). The L references the left side, a path coming into the vertex. B is the base contained in the vertex itself. R is an outgoing path from the vertex (Figure 2.5). More conceptually, contexts can be seen as paths which pass through a given vertex. Because these paths are linear and pass through vertices containing characters, the contexts can be treated as text strings. There are two concrete examples of approaches presented in the article: The *general left-right exact match mapping scheme* and the *central exact match mapping scheme*. The keywords left-right and central refer to how a vertex defines its contexts based on the surroundings. The former examples define separate contexts for incoming and outcoming paths whereas the latter defines the vertex as the center of a path where the differences of the lengths of the two contexts are minimized. The *balanced central exact match mapping scheme* is a special case of the centralized scheme where both contexts are the same length, and the vertice thus is the center of a k-mer. This is a concept closely related to de Bruijn graphs.

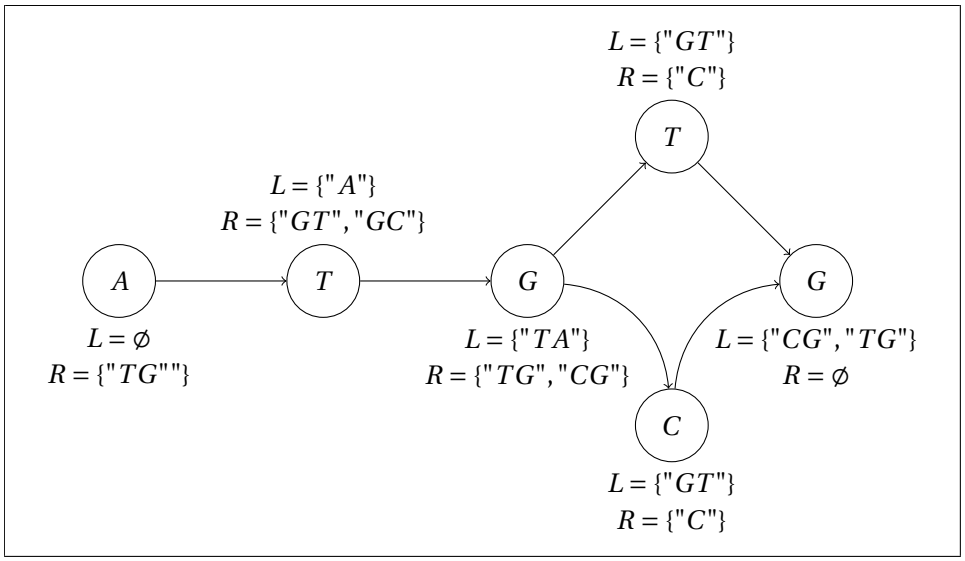


Figure 2.5: A sequence graph with contexts of length 2 explicitly shown next to the vertices. All contexts are seen in the direction of traversal away from the vertex

Both of the examples use the word *exact* in their definitions. The term refers to the fact that every context is *unique* to a single vertex, meaning every possible context either maps unambiguously to a single vertex or does not map at all. As the graphs have the possibility of branching, a vertex can have several contexts contained in a *context set*. Because every context is unique, a collection of such will also be unique. The results is a two-way unique mapping schema. This is an even stronger notion of mappability than positions in strings, as a character of a string does not necessarily map uniquely back to its position. Being this precise in the definition has a drawback: If a vertex does not have a unique context set, it is no longer mappable. In spite of this, the context-based approach presents a precise and efficient solution to the mapping problem. This is knowledge we will bring with us when we move on to considering the more complex alignment problem.

2.3.3 Alignment

As previously discussed, alignment of text strings has for some time been considered solved. We let the two strings represent each their dimension in a two-dimensional space and search for a path through the space which yields an optimal score. When one of the strings is replaced with a graph, a simple two-dimensional representation is no longer sufficient. The "3 steps before" or "11 steps after" relationship found in strings is no longer as easily derivable. A solution to this problem can be to imagine alignments against graphs like alignments against sequences in a database: There exist several possible sequences which can be aligned two-dimensionally, find the one yielding the highest score. But, unlike individual sequences in a database, the paths through a graph can have overlapping regions. Creating all possible paths would result in an exponential number of possibilities which does not necessarily portray a fair picture of the underlying structure.

Dynamic programming on graphs

The article "Multiple sequence alignment using partial order graphs" [20] proposes a direct adaptation of the regular two-dimensional dynamic programming solution for graphs through the *Partial Order Multiple Sequence alignment* (PO-MSA) algorithm. Every vertex contains a one-dimensional array representing the string which is to be aligned. Just like a single array-index in the edit distance problem, the vertex looks at smaller subproblems to decide what the values of the array should be. However, because this is a graph and not a string, it is no longer sufficient to look up the preceding index. The vertex has to look at every preceding vertex as a single instance of the two-dimensional problem, to determine which of the incoming vertices represents the linear path which presents the highest score. After filling out every index i of the array in the vertex v in this fashion, the array represents the highest score possible for the substring $S[0:i]$ for all paths ending in v.

Using an approach which is this closely related to the known approaches for regular string alignment has its advantages. Alignments and scores are verifiable through existing tools, and the principle of optimality is contained through the dynamic programming principles. Techniques for handling the different types of alignments, for instance local or global, can be inherited from the domain of strings. The algorithm is, however, a crude adaptation and thus susceptible to the inherent complexity of graphs.

Context-based alignment

In the article "Canonical, Stable, General Mapping using Context Schemes"[33] the previously mentioned concept of context-based mapping is used for aligning entire strings. The algorithm works by identifying substrings of the input string which align uniquely to a context in the reference. Overlapping contexts are combined into longer *Maximal Unique Substrings* (MUMs) which uniquely align to a region of the reference. Finally, the aligned substrings are combined in chains into β-synteny blocks, paths along the graph where exactly β mismatches are allowed between the uniquely mapped elements. Any remaining bases are mapped *on credit*, for instance as a regular graph search through the region represented by the gap between the end and start of to consecutive uniquely mapped subsequences. The conceptual idea is that any string mapping to a region of the graph should share a number of unique paths, which can be combined into a larger result. The authors name their heuristical approach the $\alpha - \beta$-*Natural Context-Driven Mapping Scheme*. The introduction of the α and β variables allows for a regulation of the strictness of uniqueness and presents a powerful approach for alignment against complex reference structures. However, it is still a heuristic; based on a non-tautological assumption for the input data.

Chapter 3

The algorithm “Fuzzy context-based search”

In this chapter we introduce the algorithm “Fuzzy context-based search” ("fuzzy search" or "fuzzy" for short) as a solution to the problem of aligning text strings against graph based reference genomes. In order to do this we will first present formal definitions of the elements and structures involved as well as the problem itself. The following description of the algorithm will be a conceptual overview where the motivation behind the steps taken is also described. A more detailed introduction to a precise implementation of the algorithm will follow in the succeeding chapter, in which space and time complexity will also be discussed. Due to the abstract nature of this chapter the reader is advised to use the coming chapter as a reference whenever needed. The two have corresponding sections; the latter contain exact details and concrete examples. There can also be value in looking up the visualizations of actual runtime examples shown in Chapter 5.

To avoid ambiguity when dealing with already existing concepts, the terms which are defined are given problem-specific names. For several of the terms, there also follows a shorthand notation behind the original name in the definition title. Whenever these shorthand names are used in the subsequent explanatory sections, we refer exclusively to the definitions given in this thesis.

3.1 The graphs

The graphs used as reference genome graphs will be built iteratively by starting out with an empty graph and sequentially merging in input sequences aligned against the existing structure. How the sequences are merged, and thus what the graphs look like, are decided entirely through the alignment procedure, which in part relies on the scoring schema. This first section is dedicated to precisely defining the involved graphs through definitions of their constituents.

Definition 1 (Graph based reference genome (Graph))
A pair $G = \{V, E\}$ where V is a set of vertices and E is a set of edges. $|G|$ denotes the number of vertices in G.

The involved graphs will be sequence graphs where every vertex correspond to a single nucleotide from a one or more input sequences used in building the graph. Whether the vertex originates from a single or several sequences is based on whether any new bases has been mapped, and consequently merged, into the vertex. In addition to the nucleotide, the vertices will contain an index which is unique to the graph.

Definition 2 (Graph genome vertex (Vertex))
A pair $v = \{b, i\}$ where $b \in \{A, C, T, G\}$ and i is a unique index. The vertice at index i is often referred to as v_i. The notation $b(v_i)$ is used to reference the first element in the pair (the nucleotide).

Every graph G will have two special vertices $s_G = \{s, 0\}$ and $t_G = \{e, -1\}$ which represents unique start and end vertices. These are the only two vertices present in an empty graph before any sequences have been merged in.

The edges in the graph model the relationships between the vertices and thus the relationships between the elements of the input sequences. Every edge has its origin from a consecutive pair of nucleotides in one or more input sequences.

Definition 3 (Graph genome edge (Edge))
An ordered pair $e = \{i_s, i_e\}$ where both elements are indexes for vertices.

There exists no information storing the origin of an edge, or whether an edge originates from one or more input sequences, and all edges are thus seen as equally probable when aligning a sequence. A sequence of vertices where there exists an edge for every pair of consecutive vertices is called a *path*. The introduction of paths is our way of capturing the combination of several individual characters into a text string in the domain of our graphs.

Definition 4 (Graph genome path (Path))
A list P of indexes such that for all consecutive pairs $(p_x, p_{x+1}) \in P$, where p_n denotes the n-th element of the list, there exists an edge $e = \{p_x, p_{x+1}\}$. The notation p_{-1} denotes the last element in the list. The length of P, $l(P)$, is equal to the number of indexes in the list. We define the distance $d(P)$ between p_0 and p_{-1} as $l(P) - 2$.

Corollary 1 (Distance between neighbours)
Every edge e is also a path P with $d(P) = 0$.

Paths spanning the entire length of a graph G, from s_G to t_G, are named full paths. Every input sequence used to build the graph has a corresponding full path. In our definition, this applies to every sequence which is aligned against and merged into the graph[1].

[1]This is a design choice further discussed in section 4.5

Definition 5 (Full path)
A path P through a graph G where $p_0 = 0$ and $p_{-1} = -1$

There is no correspondence the other way, meaning there can exist full paths which do not originate from a single input sequence. An example of this can be seen in figure 3.1 where a reference graph made from three sequences has nine valid full paths.

When aligning regular text strings against each other, the introduction of gaps is a key element. We translate this concept to the graph domain through the introduction of *incomplete paths*.

Definition 6 (Incomplete path)
A list $P*$ of indexes such that for all consecutive pairs $(p*_x, p*_{x+1}) \in P*$ there exists a path P such that $p_0 = p*_x$ and $p_{-1} = p*_{x+1}$.

Conceptually incomplete paths can be seen as regular paths where some of the vertices are removed to reflect gaps. An example of an incomplete path seen in figure 3.1 is $[1,2,4,5]$. We can score an incomplete path by looking solely at the gaps present and avoiding mapping scores for the nucleotides contained in the vertices to produce a *path score*.

Definition 7 (Path score)
The total score of all gaps present in an incomplete path $P*$ according to a scoring schema. Referenced by $pathScore(P*)$.

In an incomplete path, there exist two possible relationships between consecutive elements: Either they are neighbours, and there exists an edge between them, or they are not neighbours and are at the beginning and end of a path. Because the edges are also paths with a distance of 0 and are thus not penalized, the path score of an incomplete path can be found by summarizing gap penalties for gaps between every pair of consecutive vertices:

$$pathScore(P*) = \Sigma_{i=0}^{|P*|-2} gapPenalty(distance(p*_i, p*_{i+1})) \qquad (3.1)$$

where $distance(x, y)$ denotes the distance of the shortest path P where $P_0 = x$ and $P_{-1} = y$. If we continue with the example incomplete path $[1,2,4,5]$ from the figure, we can see one pair on consecutive vertices which are not neighbours: $(2,4)$. We can see that the only path between them, the path $[2,3,4]$, has a distance of 1.

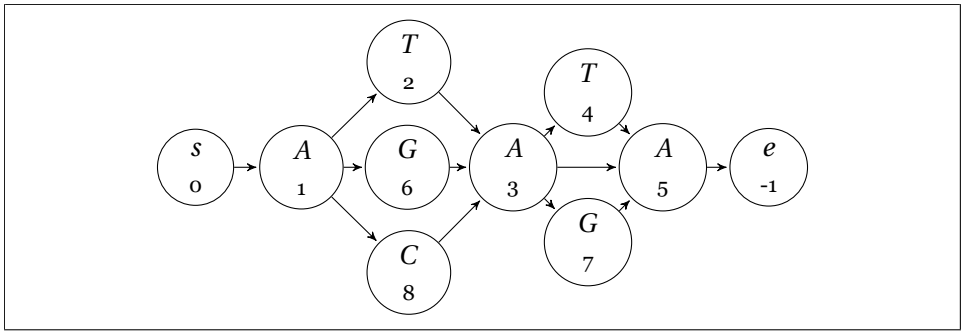


Figure 3.1: An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA"

3.2 The alignment problem

Defining the graphs mean we have a formal notion of one half of the input data for the alignment problem as discussed in section 2.3.2. We now define the other half: The *input sequences*.

Definition 8 (Input sequence)
A string s over the alphabet $\{A, C, T, G\}$. The length of the string is given by $|s|$. The individual character on position $0 <= x < |s|$ is referenced by s_x. A substring spanning the characters from x to y is denoted $s_{x:y}$

Both in defining the graph vertices and the input strings we put a limitation on the legal characters by defining their alphabets. This is done to stick with the concept of genetic information. The approach is, however, general enough to handle arbitrarily large and complex alphabets, as long as a sufficient scoring schema is provided.

Once we have a clear definition of a graph G and an input sequence s we can specify what an *alignment* between the two should look like, representing a model of the relationship between them. In order to achieve this goal, the alignments should provide relations between the smallest constituents of the two input structures, the vertices of the graph and the characters of the string, in a way such that the internal structures of the two are reflected against each other. We can model an alignment as a special variant of an incomplete path, which allows for *unmapped elements*. These elements are recognized as elements of s which is mapped to 0, the index of the start-vertice, and thus always an invalid mapping. The remaining elements of s are mapped to indexes of valid vertices of G which form an incomplete path $P*$. Moving forward through the individual positions s_x which are mapped corresponds to traversing $P*$.

Definition 9 (Alignment)
Given a graph G and a string s, an alignment A is an ordered list of length $|s|$ such that every element $a_x \in A$ is either 0 or the index for a valid vertice

of G such that for every consecutive pair of valid indexes (a_n, a_m) there exists a path P where $p_0 = a_n$ and $p_{-1} = a_m$. A 0 represents an unmapped character in s.

When we have defined the alignments we can start scoring them. The scoring happens according to a scoring schema and should be the sum of three different scores:

1. The mapping scores of the mapped elements
2. The gap penalties for gaps in the graph, represented by the path score of the incomplete path P*
3. The gap penalties for gaps in the string, represented by unmapped positions

The first two can be looked up through the standard functionality provided by the scoring schema and equation 3.1. The last can be found by summing up the gap penalties for all the gaps in the input sequence. A gap in the input sequence can be identified by a continuous subsequence $A*_{x:y} \in A$ where every element is unmapped. An important aspect is that every unmapped element should only be considered part of exactly one gap. We cover this by only considering *maximal unmapped subsequences*

Definition 10 (Maximal unmapped subsequence)
A subsequence $A*_{x:y} \in A$, such that $a = 0$ for every $a* \in A*$ and x is either 0 or $a_{x-1} \neq 0$ and y is either $|s| - 1$ or $a_{y+1} \neq 0$.*

The gap penalties for gaps in the string is then defined as

$$stringGap(A) = \Sigma_{A* \in A_U} gapPenalty(|A*|) \tag{3.2}$$

where A_U is the set of maximal unmapped subsequences in A. Once we have clear definitions of the three elements we can define the score itself:

Definition 11 (Alignment score)
Given a sequence s, a graph G and an alignment A, the score produced by combining mapping scores for the pairs $\{b(v_{a_x}), s_x\}$ for $0 <= x < |s|$ where $a_x \neq 0$ with the path score for the incomplete path provided by consecutive mapped indexes of A and the gap penalties for A_U. We reference this score by φ_A.

We can then easily define our graph based adaptation of the alignment problem[2]:

Definition 12 (The optimal alignment score problem)
For any pair $\{G, s\}$, where G is a graph and s is an input sequence, find one of the alignments A which produces the highest possible alignment score.

Notice that the definition only calls for finding one of the alignments which produce the highest possible score. This is done to simplify the conceptual explanations of the algorithm. Implementation-wise this can trivially be

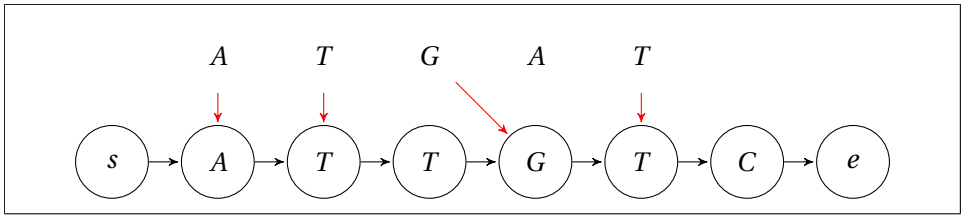


Figure 3.2: A visualization of an example alignment of the string "ATGAA" against a reference genome made from the string "ATTGTC". The actual alignment is visualized through red arrows pointing from the characters of the string to the vertex they are mapped to. We see a gap in the incomplete path between the second and third vertex and that the fourth element of the string is unmapped

changed to finding all optimal alignments. The necessary adjustments are discussed as a part of the succeeding chapter in section 4.1.3.

Additionally, we have defined a bounded version of the problem, which we call *The bounded optimal alignment score problem*. This second version also considers a score threshold value T and deems a string *unalignable* if the optimal alignment produces a score lower than T.

Definition 13 (The bounded optimal alignment score problem)
Given a triplet $\{G, s, T\}$ *where* G *and* s *are as before, and* T *is a real number, find the alignment* A *which produces the highest alignment score, if and only if the alignment score for* A *is higher than* T*. If no such alignment exists,* s *is classified as unalignable.*

Defining a bounded adaptation of the problem is obviously done in order to reduce the computational complexity, but it also present a powerful notion of control to the model: We can choose the degree of similarity required for substructures to be considered equal. This simplifies the concept of equality to a classification problem where the border between the two classes can be easily manipulated through the threshold variable.

At this point, we want to point out a distinction which does not become clear through the definition of the problem. The main goal of the algorithm is aligning short reads against a large reference. The problem definition does not in any way concern itself with the sequence length. We, therefore, assume our approach is used as a basis for building the graphs, by aligning longer sequences and merging based on this alignment. The length of s does in no way interfere with the validity of the approach, but can be of interest to the user when considering the complexity analysis we provide underway.

[2]The regular alignment problem for strings as describe in 2.2.4

3.3 "Fuzzy context-based search"

Having properly defined the problem, we now present our algorithm as a proposed solution. The algorithm consists of two distinct subproblems which are solved in consecutive steps:

1. Create a candidate graph G' for an input triplet $\{G, s, T\}$
2. Search G' for an optimal alignment

Both the motivation behind each step and the conceptual approach for solving the subproblem will be explained in its corresponding subsection. In addition to the three involved components given as input parameters, the algorithm assumes a predefined scoring schema.

In presenting the algorithm, we introduce a new variable λ. λ represents the *error margin* allowed in an alignment and is computed by taking the difference between the highest possible alignment score for s and the scoring threshold T. In order to compute the highest possible alignment score for any string, we put a bound on our scoring schemas by introducing *consistent scoring schemas*. A scoring schema is consistent if the highest possible alignment score for any string s is achieved by aligning the string against itself. This presents us with an easy computation for finding the score we need. Introducing λ gives us the opportunity to do strict pruning throughout the entire alignment process: Any alignment which contains a single element, be it a gap or a sequence of mappings, which is penalized more than λ compared to the corresponding element in an optimal alignment can never have a total alignment score higher than T (A compact proof can be found in Appendix A).

3.3.1 Constructing the candidate graph

The motivation behind building an entirely new graph is the realization that whenever reads are mapped against a reference genome, the read is typically vastly shorter than the reference. We can, therefore, do a *horizontal pruning* where we determine which sections along the horizontal axis of the graphs are interesting for the alignment. The same argument can be made for extremely complex graphs, where only a small number of the branches are relevant, in an operation we have called a *vertical pruning*. The result of the pruning should be a new graph G' with a vertice set V' and an edge set E'.

We first define V' as a subset of the original vertex set V. To guarantee optimality we put a restriction on V':

1. Every $v_x \in V$ should be in V' if there exists an optimal alignment A with an alignment score $\varphi_A \geq T$ which contains the index x

Through the definition of the alignments, we know they are ordered and that the indexed elements refer to the vertices which map to a specific

position in the string. We can use this knowledge to more specifically define V' as an ordered set of sets V'_x where every indexed set is related to the corresponding position in the alignment:

1. Every $v_y \in V$ should be in V'_x if there exists an optimal alignment A with an alignment score $\varphi_A \geq T$ where $a_x = y$

This is a restriction which is strictly enforced throughout the algorithm, to continue ensuring the optimal solution exists as a possibility. We formulate a second restriction, to reduce the number of vertices we identify as not interesting for the final alignment:

2. Every $v_y \in V$ which is not referenced by a_x in any optimal alignment should not be in V'_x

If we manage to create V' from these two restrictions, we can guarantee a vertice set where every element of every optimal alignment is still present, and all excesses vertices are dropped. However, finding these vertices requires knowledge of every alignment A of every string s for every threshold T, a number of possibilities which quickly become infeasible. To make the operation more tractable, we identify the second restriction as being related solely to the computational complexity, which means it does not have to be strictly enforced. We can thus relax it without interfering with the principle of optimality:

2. Every vertice $v_y \in V$ should be in V'_x for every $0 <= x < |s|$.

This is a complete relaxation and puts every vertice $v \in V$ in every subset of V'. The resulting parenting candidate vertice set V' is a set far greater than V which is obviously suboptimal for the following search. These two cases represent the two extremes on the scale of how strictly we enforce the second restriction, and they both represent problems: Either the search is too complex, or the result is too inaccurate. We can let the second restriction be an informal description of a search for an optimal middle ground between the two:

2. Every subset V'_x should be *as small as possible, without the search growing too complex*

The rest of this section will describe our method for approximating this middle ground without interfering with the strictly enforced first restriction.

We let a vertice v be a *candidate vertex* for index i if it is a part of the *candidate set* V'_i. In order to find candidate vertices we apply *fuzzyness* to the context-based mapping schema proposed by Paten et al[3]. We say a vertex is a candidate vertex for an index if it has a context which is similar enough to the context of the corresponding position s_i in s. The vagueness of "similar enough" is controlled through the fuzzyness, which again is controlled through the error margin parameter λ. The contexts of the vertex

[3]Explained in detail in section 2.3.2

represent the paths passing through it, and because we know that if a context is penalized more than λ compared to the maximal possible score the context can never be a part of a longer incomplete path with a total score higher than T. Thus, more formally, for every index $0 <= i < |s|$ we put v_x in V'_i if and only if the context set $c(v_x)$ of v_x contains a context which can be aligned against $c(s_i)$ with a score higher than T_c. When we refer to the context set $c(s_n)$ for elements of the string, we simply mean the only linear context possible, a substring of s surrounding the character in position n. T_c is a *context threshold score* and is computed by taking the max possible score for a context in s and subtract λ.

After deciding which vertices make up G' we need to decide how we combine them, through the edge set E'. Because the subsets of candidate vertices follow a natural ordering, there is already defined a direction in the graph. Every vertex of every candidate set V'_i should have an incoming edge from every vertex in the preceding candidate set V'_{i-1} to account for this directionality. Because we allow gaps in our alignments we have to extend the number of steps a vertex looks backward for possible paths: Every vertex in every candidate set V'_i should have an incoming edge from every vertex in *every* preceding candidate set V'_j, where $0 <= j < i$. These edges represent the relationships between the elements of the string. We also want to represent the relationships between the vertices in the graph they originate from. This is done through the introduction of *weighted edges*:

Definition 14 (Graph genome weighted edge (Weighted edge))
A triplet $e' = \{i_s, i_e, w\}$ where the two first elements are indexes for vertices in V' and w is a non-negative integer. We let w denote the distance of the shortest path P with $P_0 = i_s$ and $P_{-1} = i_e$

Corollary 2 (Weighted edges for neighbours)
For every edge $e = \{i_s, i_e\} \in E$ where $v_{i_s} \in V'_x, v_{i_e} \in V'_y$ and $x < y$ there exists a weighted edge $e' = \{i_s, i_e, 0\} \in E'$

These weights can be found through a regular graph search in G. If no distance is found, which will happen when i_e precedes i_s or they are on separate branches, we let the value be ∞. At this point we have a complete candidate graph G', but it is conceptually still very complex. Every vertex is connected to every preceding vertex. To find the weights of these edges we need to do graph searches for every possible pair of vertices. However, we still know we are not interested in alignments which have an alignment score $\varphi_A < T$. We can thus limit the edges to only representing gaps that are traversable without being penalized more than λ. This creates an upper bound both on how far back in the candidate sets a vertice looks for incoming paths, and, more importantly, the complexity of the individual graph searches done in G to find distances.

3.3.2 Searching the newly formed graph

We have built G' in a specific way to guarantee the optimal alignments still exist, which means the next step is finding them. Searching for an alignment means combining vertices, representing bases, into a path representing a string. This linear sequence can be aligned against the input sequence with regular string alignment tools and the scores are therefore easily verifiable.

In order to continue securing optimal results, the algorithm does the search using exhaustive dynamic programming. The search algorithm is conceptually very similar to PO-MSA[4], except the roles are switched around: Instead of searching through the reference graph with an input string we are searching through the indices of the string with the vertices from the candidate graph as our input. When we dynamically compute scores we are still doing the same thing as a regular PO-MSA, letting a candidate vertice v_x in a candidate set V'_i be an intersection at position x, i in a two-dimensional space where the dimensions represent the string and the path. We let an individual score identified by x, i be the highest possible score for aligning the substring $s_{0:i}$ against a path ending in v_x. In this way, we can find the highest possible score for the entire alignment in the highest scoring vertex in the last candidate set.

The base cases of the dynamic programming are the candidate vertices in the first candidate set, $v_x \in V'_0$. We initialize these scores to $mappingScore(b(v_x), s_0)$, which is equivalent to aligning them against the substring containing exactly the first character of the string. During the following bottom-up procedure, we will be faced with another set of base cases: Vertices which have no incoming edges. If the vertices are reachable by gapping over the preceding indexes of the string without the gap penalty exceeding λ we initialize them to their mapping score combined with the gap penalty. In all other cases, we set the score to an arbitrary low value which yields any alignment starting with the vertex a score lower than T. This represents the fact that we no longer consider them as viable candidates for an optimal alignment.

The recurrence relation of the dynamic programming algorithm is concerned with setting the score for any vertice/index pair which is not a base case. The score for these candidate vertices $v_x \in V'_i$ are set by looking at all incoming weighted edges, find the one yielding the highest score and add $mappingScore(b(v_x), s_y)$. The score for an edge is found by taking the score for the vertice v_{i_s} and adding the gap penalties corresponding to traversing the edge. There are two gap penalties related to the edge: one penalty for the distance represented by the weight w and one penalty for jumping from index i to index j. However, all edges traversed in the final alignment will only be penalized for one of them. We know this because whenever there exists an alternative with only one gap, this will be prioritized due to a lower

[4]The DP algorithm developed in [20] presented in 2.3.3

gap penalty. Whenever there does not exist such an alternative, this means the candidate vertex which "should" have existed is not a member of any contexts scoring high enough, meaning the path can never be part of an alignment with a score higher than T.

When the scores have been computed for every candidate vertex, we can start looking for the highest score, which represents the alignment score for the optimal alignments. We will find this score as a score for one of the vertices in the candidate set corresponding to the last index of the string. At this point we just have to backtrack the procedure which lead to the score to find the actual alignment, which is guaranteed to be one of the optimal alignments. If we find no score higher than the threshold T we simply deem the string as unalignable.

3.4 A heuristical modification

The second step of the algorithm does an exhaustive search over a bounded area of the solution space. This is an area which is defined through the first step searching for candidate vertices: The first restriction assures it still contains all optimal solutions while the second seeks to minimize it as much as possible. We classify a string as unalignable if we can no longer guarantee the lower bound set by the first restriction have been preserved. We recognize this through the realization that we need to lower the threshold T in order to produce an alignment. This means we should also have lowered T in the pruning done in the first step of the algorithm, to ensure no candidate vertices are missing. There is no longer an equivalence between the best alignment in the candidate graph and the original graph, which is a necessity to guarantee optimal results.

The boundary which is put on the solution space can be expanded through increasing λ. This does, however, come at the cost of computational complexity: A more spacious boundary takes more time to find and leaves a larger area we have to search through. To increase feasibility we can sacrifice the guarantee for optimality to avoid a growth in complexity. We know the candidate vertices stem from contexts which align sufficiently good against substrings of s, and we have found the optimal path through them. Instead of classifying the string as unalignable, the heuristical version of the algorithm returns the path which is found without doing any validation to confirm whether or not it is optimal.

Chapter 4

Implementation

In this section we will present the implementation of our algorithm which can be found in the *GraphGenome* tool (Appendix C). The algorithm will be coupled by an explicit example found in figures 4.1-4.3 and table 4.1. Throughout this chapter, and in the example case, the scoring schema is assumed to be what we have called the *negated edit distance* schema. The schema takes its values from the regular edit distance problem[1] and negates them. The reason for the negation is that the tool is implemented for generalized scoring schemas which attempt to maximize alignment scores. Otherwise, the scoring schema is chosen due to its intuitive nature: A final score of $-x$ means there are exactly x differences. The scoring schema is also practical for doing complexity analysis: A gap which is penalized by y means traversing exactly y vertices or indexes.

The chapter is divided into two main sections. The first explains the implementation of the algorithm corresponding to the previous chapter. The second is a brief overview of how the tool merges sequences into the graph after they have been aligned. This section is included to better provide an intuition as to how the graphs are built and thus what readers can expect when seeing the results in the suceeding chapters and using the tool themselves. Additionally, after the two main sections describing the steps of the algorithm, we will briefly present some modifications of the implementation which are referenced in the remaining chapters of the thesis.

4.1 Aligning sequences

The alignment process consists of the two steps described in the previous chapter, which each have their corresponding subsection. In addition to these, the tool needs to do a precomputation of the graph in order to build a searchable index. This is not counted as a step in the alignment process as the precomputation is dependant only on the graph and the index is thus reusable for several alignments.

[1] match=0, mismatch=1, gap opening and extension penalty=1

4.1.1 Building the index

There are two data structures needed for aligning a string against the graph: A suffix trie for left contexts and a suffix trie for right contexts. Before either of the two are built the algorithm needs to decide a length for the contexts. In the tool there are two ways of setting the context length: A user given parameter or an approximation based on the probability of sharing contexts (Appendix D). The length of the contexts does not impact the quality of the alignments found by the algorithm (Shown in the proof in Appendix A) but will have an impact on the runtime (Calculations can be found in Appendix B).

When a context length $|c|$ is set, the algorithm can start building the index. Two sets of strings, a left context set and a right context set, are generated for every vertice in the graph G. Because the algorithms for the two are equal, aside from the starting point and the direction of the traversal, the following explanation only describes one of them.

The generation of the left contexts start by adding an empty context ϵ to the context set $c(v_i)$ for every vertice v_i which is a neighbour to s_G, and inserting these vertices in a regular FIFO-queue. The contexts are stored in an individual array of sets of strings where the indexes correspond to the indexes of the vertices. The algorithm marks s_G as finished in a boolean table and starts iterating over the elements of the queue.

When a vertice v_x is popped from the top of the queue the first operation consists of checking whether the context set of the vertice is done. A context set is complete if every vertice in the incoming neighbour set $n_i(v_x)$ of the vertice is marked as finished. If the context set is not complete the vertice reinserts itself at the end of the queue. In the opposite case, when a vertice is deemed as ready, it starts generating contexts for its outgoing neighbours $n_o(v_x)$. The vertice takes every context c belonging to its own context set $c(v_x)$ and creates a new context c' by prepending the base $b(v_x)$ to the string. If necessary, when the length of a new context exceeds $|c|$, the trailing character of the string is also removed. Each of these newly created contexts are added to the context sets of every outgoing neighbour $v_y \in n_o(v_x)$ and v_y is added to the queue. Every vertice should be enqueued at most once to avoid an exponential growth in queue size. This is enforced through efficient lookup of currently enqueued vertices in a hash set. The final step for the vertice is marking itself as finished.

In order to avoid making the end vertice t_G mappable the algorithm strictly puts it back at the end of the queue whenever it is seen. When the queue contains a single element, and this element is t_G, the algorithm halts. At this point every vertice along every path leading up to t_G has generated its context, and as we know every vertice stems from an input sequence and every input sequence ends in t_G we know every vertice has been visited. s_G

```
while queue does not consist of t_G do
    current = queue.pop
    if all incoming neighbours are not done then
        enqueue current;
        continue;
    end
    for every context c do
        for all outgoing neighbours do
            suffixes[outgoing.index].put(b(current) + c);
            if outgoing not in queue then
                enqueue outgoing;
            end
        end
    end
    finished[current.index] = True;
end
```

Algorithm 1: The loop which generates left contexts for a graph

and t_G swaps places as start and end-vertices, the definitions of what is an incoming and an outgoing neighbour is switched around, and the algorithm starts again to generate right contexts.

As sets per definition do not allow duplicates, the impact of a branching occuring in the graph will fade away after exactly $|c|$ steps. At this point the difference is trimmed away (Figure 4.1), and exponentiality in the context set sizes is avoided. Unlike the method of Paten et al. [34] there are no requirements for contexts to be uniquely mappable to exactly one vertice. Because the last step of the algorithm does an exhaustive search for an incomplete path through all the candidate vertices this presents no difficulties when finding the alignment. Furthermore, dropping this precondition assures every node has two valid contexts and are thus present in both suffix trees.

After generating the two context sets for every node, the elements of each one is inserted into their corresponding suffix tree. Every suffix is stored as a key with the index of its originating node as a value (Figure 4.2). In theory every node can have $4^{|c|}$ contexts in each set, in practice a more fair approximation is $b^{|c|}$ where b is the observed branching factor for the graph. If we assume our graphs are too a large degree linear, we can assume $b \approx 1$ and approximate $b^{|c|} = 1$.[2] The current implementation uses a naive suffix tree implementation where insertion is $O(|c|)$, giving a total number of operations of $b^{|c|}|c|$ per node per context set and $2|G|b^{|c|}|c|$ for the entire graph. The generation process visits $|G|$ vertices once in each direction,

[2] Actual computations for b can be found in section 6.5

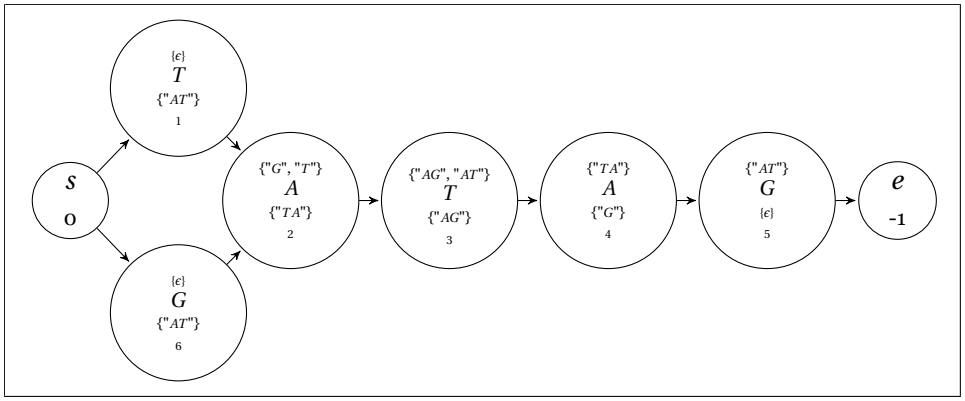


Figure 4.1: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

which means the entire index can be built in $O(|G||c|)$[3].

4.1.2 Generating the candidate graph

Unlike the index built in the previous step, the candidate graph G' is a function of both the original graph G, the input sequence s and the threshold T. For every character $s_x \in s$ a left-context string and a right-context string is generated by looking at the $|c|$ surrounding characters. The two context strings are used as a basis for a fuzzy search in its corresponding suffix trie, a recursive function based on PO-MSA. The root node is supplied with a one-dimensional scoring array $scores$ corresponding to the context string c, which is initialized with all zeroes. Then, for every child, a new scoring array $scores_b$ is computed by regular edit distance rules: For each index i take the maximal score for either a gap in the graph, a gap in the string or matching the character c_i with the character b contained in the child vertex:

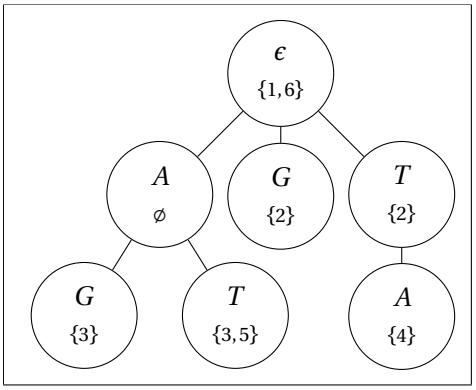


Figure 4.2: The left suffix tree corresponding to the graph in 4.1

$$scores_b[i] = max \begin{cases} scores_b[i-1] + gapPenalty(1) & \texttt{\# String gap} \\ scores[i] + gapPenalty(1) & \texttt{\# Graph gap} \\ scores[i-1] + mappingScore(c_i, b) & \texttt{\# Mapping} \end{cases} \tag{4.1}$$

[3]Assuming $b^{|c|} = 1$. Explanations of simplifications done in big-O notation can be found in the bibilography[52, Chapter 2][2, Section 3.1]

An important aspect is that this procedure uses the same scoring schema as the one defined for the entire alignment. The newly created array $scores_b$ is supplemented to the same recursive function in the child corresponding to the character b. When a leaf vertex is reached the last index of the supplied scoring array corresponds to mapping the entire string c against the entire context achieved by concatenating the characters contained in the path through the tree traversed by the recursion. If the score is higher than the context threshold T_c for the given context string, every index contained in the vertex is stored as a pair on the form $\{index, score\}$. The candidate sets are implemented as maps with the index as a key, which allows us to store the index of every candidate exactly vertex once by only saving the pair which produces the highest value.

In order to also be able to look up contexts which are shorter than the contexts stored in the tree, the suffix tree search implementation has built in a concept we called *max score inheritance*. This concepts allows all suffix trie vertices at a depth greater than the length of the string which is looked up to inherit a maximum score from their parenting vertex. Doing this we can avoid deterioration of the scores as the searched contexts no longer needs to introduce gaps to align against the longer, stored contexts.

Additionally the suffix search does one more optimization. Whenever the context is short, defined in the tool as *shorter than* $|c|$, there will be a large number of matches. The algorithm has to iterate over every single one to subtract its indexes. In these cases the tool simply returns an empty set which is treated as a set containing every single index with a maximal score. But even this seemingly simple operation has pitfalls to avoid. Whenever the provided string has a length $|s| < 2 * c + 1$ the middle elements will have contexts on either side which is not searched due to their length, which provides two empty sets. To avoid this the suffix tree search has built in a *force* option, which assures at least one set of candidate vertices is always found.

In theory every leaf vertex has to be visited in order to check the score for every represented context in the tree. In practice the tree can be pruned by cutting off the search whenever the *maximal potential score* falls below the threshold T_c for the provided context. The maximal potential score for a branch is found by adding together the currently highest score in the scoring array with the maximal matching score for the remainder of the string. This reduces the number of nodes to be searched from $O(4^c)$ to $O(|c|^\lambda)$[4].

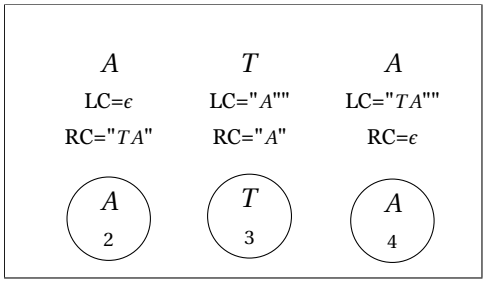
After the fuzzy search is concluded there are two maps of candidates for every index, one containing the vertices matching the left context and an equivalent for vertices matching the right context. The keysets of these maps are intersected to produce a final candidate set for the index i, where the score is created by adding together the scores from the two original

[4]See calculations in Appendix B

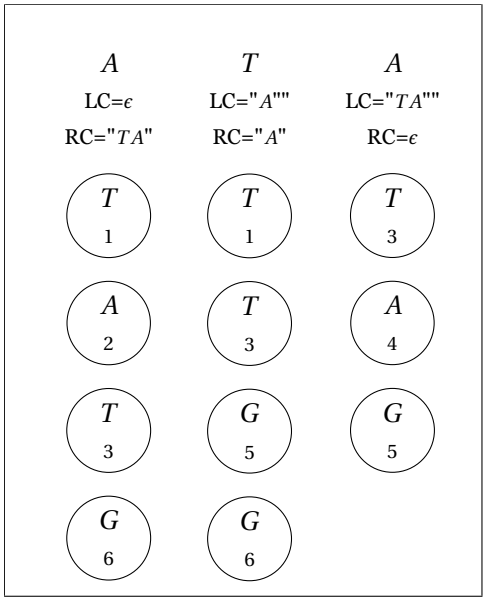
sets. During the intersection process the final set can again be pruned by removing all vertices which has a combined score that is lower than the combined threshold T_c for both contexts. If one of the sets are empty, due to pruning in the suffix tree search, we simply emulate the intersection by keeping the non-empty set. Formally we can define the new vertice set V' as an ordered list of sets V'_i for $0 \leq i < |s|$ where

$$V'_i = \{v_x | v_x \in G \wedge \exists [c \in c(v_x)](contextScore(c, c(s_i)) >= T_{c(s_i)})\} \quad (4.2)$$

where $contextScore$ is a regular 2-dimensional alignment score.



(a) $\lambda = 0$



(b) $\lambda = 1$

Figure 4.3: The resulting candidate sets for mapping the string "ATA" against the reference genome from fig. 4.1 with varying T values

Intuitively this should be the place where the edges are created in order to finish up G'. There are two relationships between the vertices which should be represented: The distance between the elements in s and the distance between the vertices in G. The first relationship is inherently contained in the indexation of the candidate sets. The second relationship is found through graph searches in G. Finding the weights of these edges is however not necessary before the involved vertices are scored in the next step, and the searches can therefore be delayed until that point. This is done in order to avoid having to use space storing data which is only necessary at a single stage of the computation.

4.1.3 Searching the candidate graph

Once we have created the graph we can start the setup for the subsequent search. We move the indexes of the vertices in the candidate sets over to a two-dimensional array $indexes$ where we let one dimension represent the indexes of the string and the second the individual vertices in the candidate set. This is done in order to have the vertices in a structure where we can reference them solely by an index.

In addition to the $indexes$ array we create a floating point array $scores$ with exactly the same dimensions, to contain the scores for the individual vertices. Additionally, in order to backtrack and find the optimal alignment, we create an equally sized $backpointer$ array. Conceptually this should store pairs of integers referencing the other indexes in the array, in the tool this is implemented using strings.

A	T	A
1	1	3
2	3	4
3	5	5
6	6	

(a) Indexes

A	T	A
-1	-2	-2
0	0	0
-1	-3	-2
-1	-3	

(b) Scores

A	T	A
-1:-1	0:0	1:1
-1:-1	0:1	1:1
-1:-1	0:2	1:1
-1:-1	0:3	

(c) Backpointers

Table 4.1: The 4 arrays used by the searching algorithm when using the candidate sets from figure 4.3

The search is initialized by looping over every node $v_x \in V'_0$ with a counter j, setting

$$indexes[0][j] = x$$
$$scores[0][j] = mappingScore(b(v_x), s_0)$$
$$backPointers[0][j] = -1:-1$$

The iteration over the elements of the set with the counter j will not be according to any ordering as the elements of the set are inherently not ordered. This is not important to us as the elements of a single candidate set have no internal relation which we want to preserve. However, from this point onward, we have given the elements an order. Although the internal ordering does not hold any value, this is important as we now know the same index in the three separate arrays reference the same vertex.

After setting the values in the easily identifiable base cases we start computing scores for the paths in our graph. The nodes $v_x \in V'_i$ for the remaining candidate sets at $1 \leq i < |s|$ are looped over with j as a counter, and $indexes[i][j]$ is set to x. For every such entry a list of pairs is made with other indexes (i', j') such that i' is a preceding index $i' < i$ and j' is another counter variable looping over $indexes[i']$. For every entry-pair $((i, j), (i', j'))$ we produce a score by the scoring function $\theta((i, j), (i', j'))$. The scoring function works by combining the score contained in the preceding entry, $scores[i'][j']$, the gap penalties, and a mapping score for the current index $mappingScore(v_{indexes[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length $distance(v_{indexes[i'][j']}, v_{indexes[i][j]})$. The computation of the last gap penalty corresponds to finding the edges, which up to this point have been without interest to the algorithm. We search for these distances by a breadth-first regular graph search which starts in the vertice $n_{indexes[i'][j']}$ and is concluded when we find $n_{indexes[i][j]}$. Whenever we search for more than λ steps without finding the target vertice, we can return the current distance multiplied by 2. When a gap penalty is computed for a gap this length the score will always be $2 * \lambda$ which means the edge can never be part

of a final alignment and is thus not interesting.

The final score stored in $scores[i][j]$ is the maximal achievable score produced by the function θ for one of the vertice pairs ending in the vertice with index $indexes[i][j]$. $backpointers[i][j]$ is set to the to the index-pair (i', j') responsible for producing this score. The recurrence formulas for the three arrays are thus:

$$\begin{aligned} indexes[i][j] &= x && n_x \in V'_i \\ scores[i][j] &= \max_{i',j'} \theta((i,j),(i',j')) && 0 \leq i' < i, 0 \leq j' < |scores[i']| \\ backPointers[i][j] &= \operatorname*{argmax}_{i',j'} \theta((i,j),(i',j')) && 0 \leq i' < i, 0 \leq j' < |scores[i']| \end{aligned} \tag{4.3}$$

where θ is a scoring function defined as:

$$\begin{aligned} \theta((x_1,y_1),(x_2,y_2)) = scores[x_2][y_2] \\ + gapPenalty(x_1 - x_2) \\ + gapPenalty(distance(n_{indexes[x_2][y_2]}, n_{indexes[x_1][y_1]})) \\ + mappingScore(b(n_{indexes[x_1][y_1]}), s_{x_1})) \end{aligned} \tag{4.4}$$

When the iteration ends we will have a score for every candidate vertice in every candidate set. We can iterate over the scores corresponding to the last candidate set, $scores[|s| - 1]$ to find the highest alignment score. The optimal alignment ends in the vertice with the index in the corresponding cell in the $indexes$ array. We can then backtrack backwards through the $backPointers$ array, storing the corresponding indexes of the vertices from the $indexes$ array along the way to produce the actual alignment. The entire operation can be done in $O(\frac{\lambda^2|s|(|c|^\lambda|G|)^2}{4^{|c|^2}})$.[5] The exponential factor $|G|^2$ is to a large degree cancelled out by $|c|$, which defaults to be algorithmically set by a function dependant on $|G|$. This leaves an operation which is heavily dependant solely on the error margin λ.

Finding all optimal alignments

There is only a small adjustment necessary to produce all optimal alignments instead of a single one, but it makes the explanation of and reasoning around the procedure considerably more complex. Briefly, the first step needed is storing a list of backpointers for every index to every preceding index which produces the same, highest score. Then, when backtracking, the algorithm needs to find every maximal score for the last candidate set. For each one of these the algorithm must start a computational branch which corresponds to each final alignment. These branches are further split up whenever faced with a backpointer consisting of more than one index. Every resulting branch corresponds to a single, equally scoring, optimal alignment.

[5] Precise calculations are done in Appendix B

There exists a third variant of the problem where only uniquely aligned reads are classified as alignable. Any read which has more than one possible optimal alignment can be classified as ambiguous and dropped. Again, there is only a small modification needed to accomplish this within the implementation. Once again we allow for branches while backtracking, by storing lists of optimal backpointers. However, this time around we cut the search whenever we discover a branching in the backtracking process. A branch represents several solutions with the same score, and the read is thus classified as unalignable.

4.2 Handling invalid threshold values

Whenever the algorithm is not able to find any alignments with a score higher than T it classifies the input string $|s|$ as unalignable. There are two scenarios where this would happen: Either the path yielding the highest score consists of a series of steps in which each individual step is considered legal (not penalized more than λ), but the combination is not good enough. The second scenario occurs when the path goes through a step in which there are no legal possibilities for traversal, which will happen when every path goes through an edge which was not found due to pruning and has been given the distance of $2 * lambda$. Both the cases are identified through not finding any high enough scores and both result in an *empty alignment*, an alignment where every element of s is unmapped.

4.3 Implementing the heuristics

Section 3.4 in the preceding chapter describe a modification which seeks to utilize the data found in these cases instead of returning an empty alignment. The first step of this process is to remove the validation and consequent rejection of alignments which do not meet the requirements of the scoring threshold. Secondly, because we now need to produce a result even when some candidate sets are empty, we need to iterate over all the scores to decide the starting point of the backtracking. This starting point should be decided as a function of the score produced by the alignment and the gap penalty for skipping eventual empty candidate sets at the end of the string. The most conceptually interesting property of the heuristical modification is how we handle gaps where the graph search for the distance between vertices was cut off before it produced a result. The normal algorithm penalizes these gaps just enough to put them under the threshold. When we discard the threshold this means they are often preferred due to the low penalty. We have no data to support the choice of penalty for these gaps, but it needs to be defined. In some cases our simple measure of distance does not even hold a meaning, for instance if the two vertices are on separate branches in the graph. Finding a useful

metric for such could provide useful in the case of *split alignments*[6], a concept which is further discussed in Chapter 7. We chose the default value of $gapPenalty(|G|)$ to produce as coherent alignments as possible. These modifications are available in the tool through the `--heuristical=true` parameter.

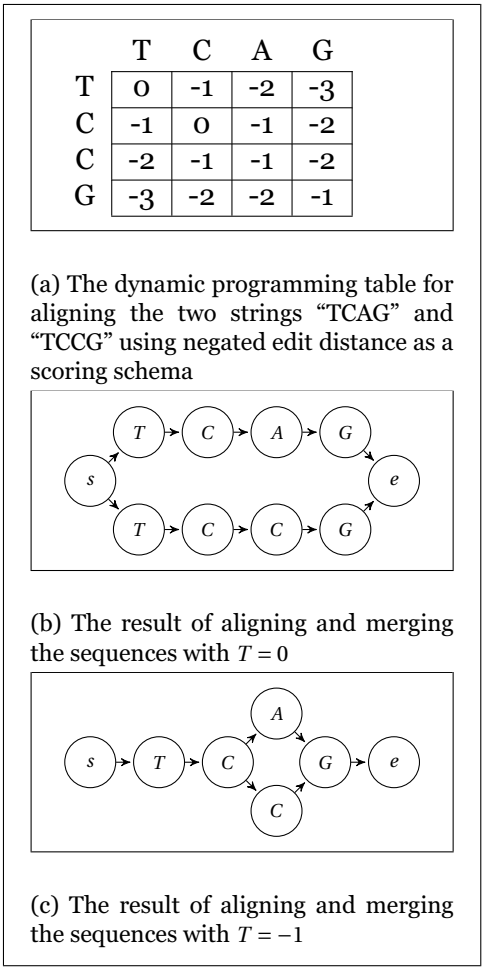
4.4 Parallelization

Of the two steps of the algorithm, one stand out as a text book case for possible parallelization. Searching for the candidate vertices for an index is done completely separate from the other indexes. An extremly trivial parallelization of this step is implemented through splitting up the indexes into separate threads, reachable through the `--parallellization=true` parameter. Further parallellization is possible as the recursive searches through the trees are only dependant on the results from the vertices higher up in the trees, there is no exchange of data between the vertices of a layer. This is not implemented in the tool to keep the proof-of-concept presentation of the approach as conceptually simple as possible.

[6] Input sequences where subsequences align to two distant regions in the graph

4.5 Merging aligned sequences

Whenever a graph is made from a set of genetic sequences there is an expectation of what the graph should look like, where there should be branches, common subsequences and so on. This is decided by how the input sequences are merged together to produce graphs. Although the merge can be seen as the key element in this process, these are desicions which are made purely by the alignment process, which in turn relies on the scoring schema and scoring threshold. Varying these two components can create a variety of graphs (Figure 4.4). Too avoid confusion we have split the two processes, letting the alignment algorithm do all the heavy lifting to keep the merge as as much of a straight forward procedure as possible.



(a) The dynamic programming table for aligning the two strings "TCAG" and "TCCG" using negated edit distance as a scoring schema

(b) The result of aligning and merging the sequences with $T = 0$

(c) The result of aligning and merging the sequences with $T = -1$

Figure 4.4: Different scoring thresholds T yields different reference graphs

Whenever we have produced an alignment A for a string s and a graph G we can do a merge, a procedure which intuitively should result in the internal structure of s being present in the merged graph $G*$. To make reasoning around the procedure as straight forward as possible, we visualize s as a special sequence graph G_s where every individual character is represented by a single vertex and every consecutive pair of characters is connected with an edge. This is a convenient representation for several reasons: For one the graph has a strict internal direction which we can use to number the vertices according to their position in the string. We let s_x denote the vertex in G_s with position x. An important note is that this is a simplified type of the previously defined graphs which does not require start and end-vertices, which means the indexation begins at 0.

The reason for this simplification is that we can use the alignment A to create an equality operator:

$$eq(s_x, v_y) = \begin{cases} True & If\, a_x = v_y \\ False & Else \end{cases} \tag{4.5}$$

When we have this way of checking equality between the two graphs we can simply let the vertice set $V*$ of the merged graph be the union of the original vertice set V and the vertice set of G_s, where every "new" vertice is given a new index when merged in. For the edges there are three cases to consider: An edge between two existing vertices, an edge from a new vertice to an existing edge (and its opposite) and an edge between two new vertices. To simplify the procedure the three latter cases can all be generalized as cases where the edge does not already exist, and needs to be created.

The implementation of this procedure is done even more smoothly. Because the string and alignment both have a direction we can move along this direction and merge or create new vertices iteratively. We let a pointer $prev$ denote the index of the previous element in the graph. Because every input sequence which is represented in the graph should have a full path we initialize this to s_G. We then iterate over the elements of the alignment. For every index i we start by creating a new vertice $curr$ if $a_i = 0$ and thus unmapped. We also create a new vertice if index is mapped, but the characters s_i and $b(v_{a_i})$ is not equal. This vertice contains the character s_i and is given a unique index x by the graph. If the index i is mapped and the characters are equal we let $curr$ be the vertice v_{a_i}. At this point we have a pair of vertices, $prev$ and $curr$, which representes consecutive characters in s and should thus have an edge between them. We create this edge by inserting $curr$ in the neighbour set of $prev$, and set $prev = curr$ to move the backpointer. When the iteration finishes we have the last vertice in the alignment contained in the backpointer and create and edge from $prev$ to t_G to finish off the full path.

There is one important aspect to be considered when using the merge procedure: Every sequence which is merged in is considered a stand-alone sequence which starts at the beginning of the graph and ends at the end. This is important because it divides the applications of the tool in two: The alignment of short reads and a combined operation of aligning and merging complete sequences. This is important to point out, because we in the previous chapter state that the algorithm is tuned for aligning reads shorter than the reference. As mentioned this does not obstruct the validity of the approach. It is however apparent that the tool is not optimized for these kinds of operations. We envision the approach as a part of a larger process, where smaller reads are aligned, combined and eventually merged. This larger process can for instance be done by overlap techniques as the results from an alignment is a set of unique IDs.

Chapter 5

Validation of the approach

In the previous two chapters we have described the approach we developed and how we chose to implement it. This chapter is concerned with showing the behaviour of the approach. When a set of sequences are combined into a graph there should be an expectation as to what the result should look like. This expectation is grounded in the underlying formalism of the approach: The definitions of the involved elements, the scoring schema used and even smaller details like the order of some operations. We increase the level of abstraction from the realm of formal details to an overview of what results can be expected from the approach.

Throughout the chapter we will be doing example runs of the tool on some input data coupled with the actual graphical output. Additionally we will present some statements regarding the state of the data structures to represent the formalization of the expectation of the results. Importantly, this is not an attempt to show whether the tool acts correctly or not, that part is covered by the next chapter. This is a chapter determined to show the reader what we classify as correct behaviour, and how this behaviour is manifested through the input/output pairs. Throughout the development and implementation process these expectations have been used as requirement specifications, realized through a test set which can be found in the github repo[1].

The scoring schema used throughout the chapter is the negated edit distance scoring schema. This is used because of the intuitive results provided by the flat scoring structure.

5.1 Test data

In order to avoid any ambiguity the test data in this chapter consists of small, hand-crafted sequences. As mentioned, the behaviour of the approach is determined by a relatively small set of formal instructions as to how it should behave in different circumstances. However, these instruc-

[1]The url is found in the presentation of the tool in Appendix C

tions can be nested. This is a step which is cruical in order to produce graphs complex enough to fairly represent genetic variation. As the nesting goes on and the structures involved grow more and more complex, finding the underlying formal rules becomes non-trivial. The subsequent tests are divided into sections, the sequences in each section are designed to represent exactly one trait of genetic data which the algorithm should handle in a specific way.

Although negated edit distance provide a good basis for intuitive results, the flat structure also presents a possible weakness: The scoring schema yields results which are susceptible to order of operations characteristics of the implementation. The test sequences provided are constructed to avoid any ambiguity concerning the order of the instructions given.

5.2 Tests

We start out with the most basic operation: Turning a sequence into a reference graph. The input sequence is given through the `--input-sequences` argument. Throughout the chapter we will first show the command, or commands, which is run:

```
./build_index.sh --input-sequences=ACGTATTAC --png=build
```

We will then show the graphical result which is stored in a png-file with the name given as a `--png` argument:

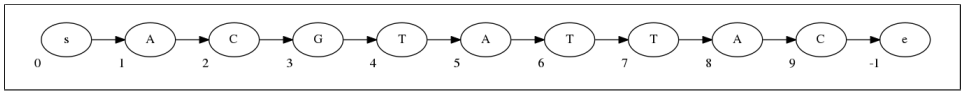


Figure 5.1: The reference graph made from the sequence "ACGTATTAC"

Finally, we provide a set of statements regarding the result:

- A reference graph made from a string s should have exactly $|s| + 2$ vertices

These statements are not exhaustive with regards to the output, as such a list would be very long and not particulary interesting. The statements provided will be what we classify as important details which follow from the inner workings of the algorithm, which in part means we omit statements which are exceedingly trivial.

The graph built in this example will provide a basis for all of the following examples. In order to provide a working setup to readers who use this chapter as an introduction to the tool the different tests are separated by their index-file, determined by the `--index` parameter, and the png filename.

Equal sequences

We move on the the most trivial alignment operation, aligning a sequence against itself:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=equal.index
./align_sequence.sh --index=equal.index
--align-sequence=ACGTATTAC --png=equal-align
./align_sequence.sh --index=equal.index
--align-sequence=ACGTATTAC --merge=true --png=equal-merge
```

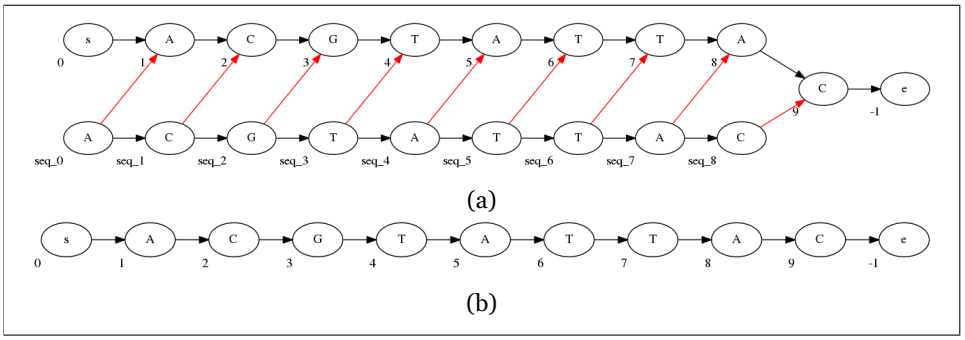


Figure 5.2: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 5.1

- Aligning a sequence against itself should provide an alignment with the indexes of continuous vertices
- Merging a sequence with itself should result in a graph with the same number of vertices as before the merge

SNPs

We let SNPs represent the first point mutation. At first we align and merge without setting the error margin λ:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=snp-no-errors.index
./align_sequence.sh --index=snp-no-errors.index
--align-sequence=ACGGATTAC --png=snp-no-errors-align
./align_sequence.sh --index=snp-no-errors.index
--align-sequence=ACGGATTAC --merge=true --png=snp-no-errors-merge
```

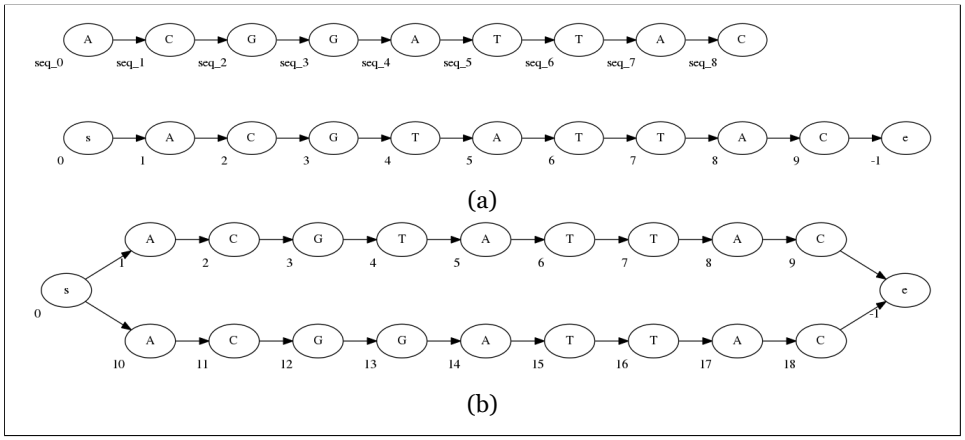


Figure 5.3: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 5.1 without setting an error margin

- Aligning a sequence with an error compared to the reference and no error margin should result in an empty alignment
- Merging in an empty alignment should result in a new full path

We then align the same sequence while allowing an error margin through the `--error-margin` parameter:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=single-snp.index
./align_sequence.sh --index=single-snp.index
--align-sequence=ACGGATTAC --error-margin=1 --png=single-snp-align
./align_sequence.sh --index=single-snp.index
--align-sequence=ACGGATTAC --error-margin=1 --merge=true
--png=single-snp-merge
```

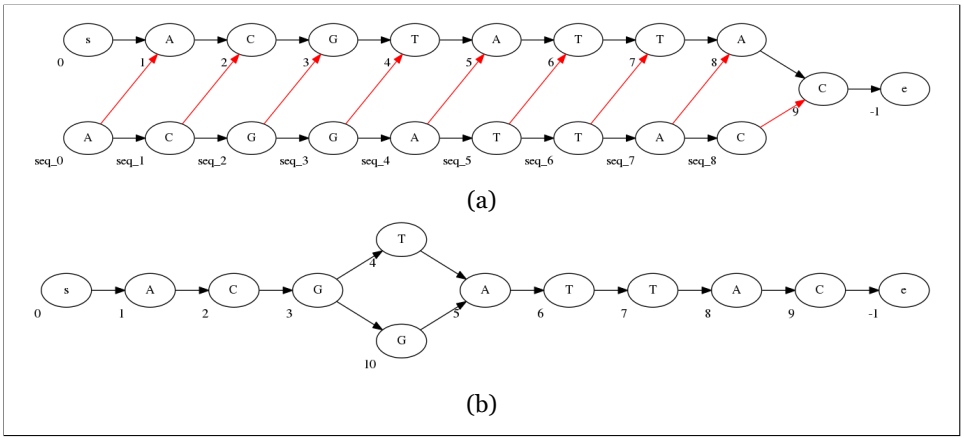


Figure 5.4: The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference seen in figure 5.1 with $\lambda = 1$

- Aligning a sequence with a SNP compared to the reference should provide an alignment with the indexes of continuous vertices
- Merging in a sequence with exactly one SNP should yield a graph with exactly one vertice more than the old graph, where exactly one vertice has one more outgoing edge and exactly one vertice has one more incoming edge.

Indels

First we test a deletion by removing the fifth character in the alignment sequence:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=deletion.index
./align_sequence.sh --index=deletion.index
--align-sequence=ACGTTTAC --error-margin=1 --png=deletion-align
./align_sequence.sh --index=deletion.index
--align-sequence=ACGTTTAC --error-margin=1 --merge=true
--png=deletion-merge
```

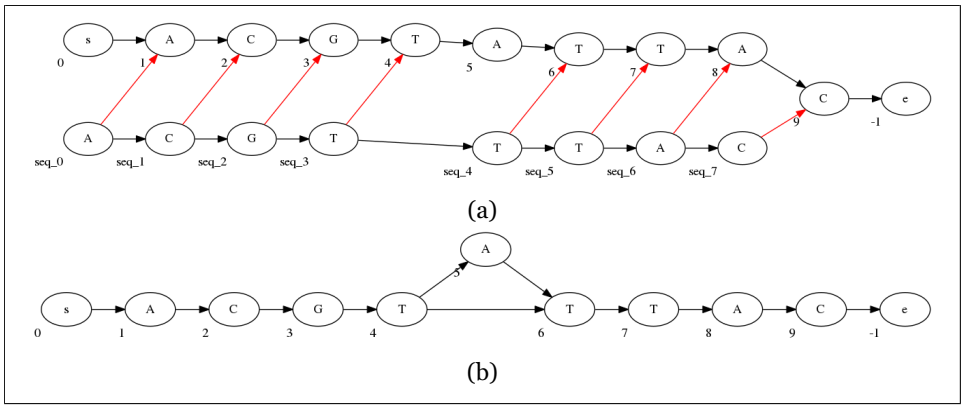


Figure 5.5: The result of aligning (a) and merging (b) the sequence "ACGTTTAC" against the reference seen in figure 5.1

- Aligning a sequence with a deletion compared to the reference should provide an alignment where exactly one pair of consecutive indexes does not represent neighbouring vertices
- Merging a sequence with a deletion should result in a graph with the same number of vertices as the old graph, but one additional edge

Secondly we test an insertion by inserting an extra 'A' after the fifth character:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=insertion.index
./align_sequence.sh --index=insertion.index
--align-sequence=ACGTAATTAC --error-margin=1 --png=insertion-align
./align_sequence.sh --index=insertion.index
--align-sequence=ACGTAATTAC --error-margin=1 --merge=true
--png=insertion-merge
```

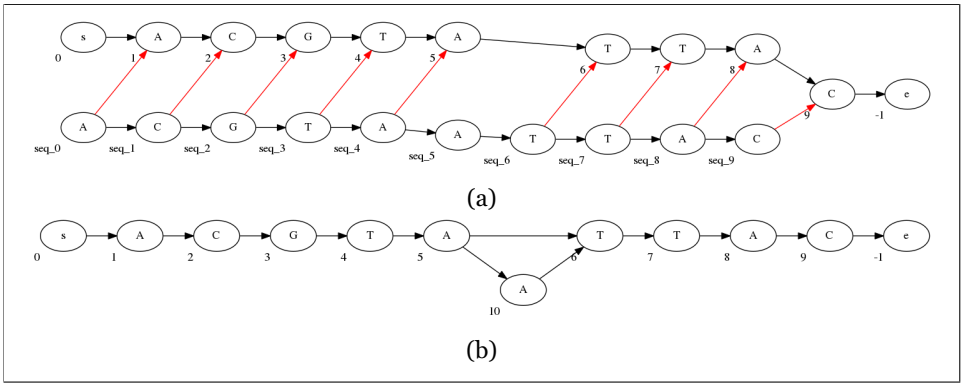


Figure 5.6: The result of aligning (a) and merging (b) the sequence "ACGTAATTAC" against the reference seen in figure 5.1

- Aligning a sequence with an insertion compared to the reference should provide an alignment of indexes of consecutive vertices split apart by exactly one unmapped index
- Merging a sequence with an insertion should result in a graph with exactly one more vertice and two more edges

Structural variation

In section 2.1.2 of the background chapter we present structural variation as mutations which occur over "larger areas of the genome". This kind of variation include larger subsequences which have been removed, inserted, moved or reversed. We can thus divide it into cases: In the case of removals we will see reads which span the sequence that is removed, represented by a large gap in corresponding path in the graph. This is a notion which does not go well with the strict mathematical notion of similarity, and is not handled well by the tool. However the approach does create data which could be useful in identifying such gaps[2]. In the case of a large insertion we will see reads which obviously do not have a counterpart in the graph. This will either result in unaligned reads or spurious matches with the most similar structure in the graph. Finding the true origin of these reads will have to be done by a larger process[3]. Large subsequences which are moved will be similar to insertions, however here we will actually align the reads to their origin instead of having spurious results. Piecing them together correctly is also a job for a combinatorical process. The latter case of reversion is simply not implemented in the tool. This is not a result of laziness: It is done strictly in order to minimize the amount of ambiguity when showing the results achieved by the approach. Neither of the cases mentioned can be formulated like the small examples in this chapter, and is thus not shown.

[2]These are the split alignments presented in 4.3
[3]Outlined in section 4.5

We will briefly treat structural variation as any variation which is not one of the point mutations depicted in the previous section. Conveniently, if we drop the actual large structural changes, these three mutations cover all the possible base cases of variation, and we can always define more complex cases as a combination of them. Because the number of possibilities grow exponentially we will only show a small fraction of cases in order to display the flexibilty of the approach.

We start out by combining an insertion with an SNP:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=structural1.index
./align_sequence.sh --index=structural1.index
--align-sequence=ACGTGGTTAC --error-margin=2 --merge=true
--png=structural1-merge
```

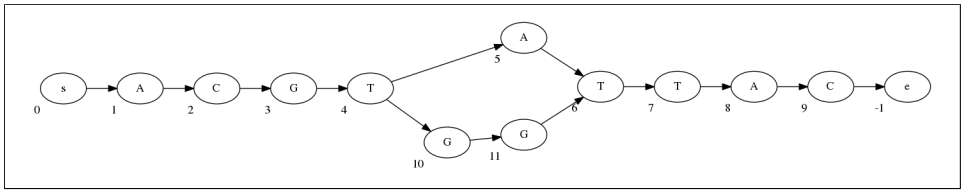


Figure 5.7: The result of merging the sequence “ACGTGGTTAC” with the reference seen in figure 5.1

- Merging in a sequence of length 10 should yield a graph with atleast one full path with length 12

When we are dealing with the "non-standard" cases we can see a statement which is less general than earlier. This is in correlation with the complexity of the input data: When the sequence consists of a combination of cases, so does the resulting graph. There is no longer a distinct relationship between the case and a rule, which can result in graphs which are not so easily deducible beforehand. The ambiguity represented by this becomes even more apparent when we align a sequence with a number of consecutive SNPs:

```
./build_index.sh --input-sequences=ACGTATTAC
--index=structural2.index
./align_sequence.sh --index=structural2.index
--align-sequence=ACGTACCTT --error-margin=4 --merge=true
--png=structural2-merge
```

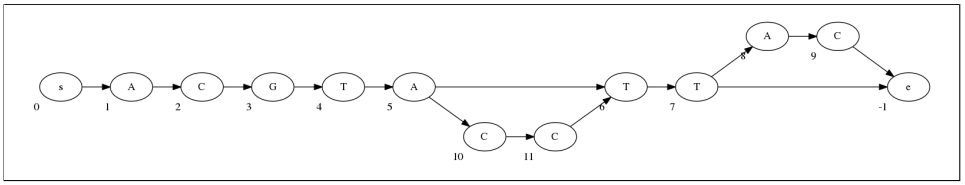


Figure 5.8: The result of merging the sequence “ACGTACCTT” with the reference seen in figure 5.1

At this point we do not achieve the result we expected. Instead of four SNPs the alignment contains two deletions and two insertions. Importantly, this is not a wrong result. The score for the alignment is exactly the same as the four SNPs we expected. This is a result of ambiguity created by the previously mentioned order of operations built in to the implementation combined with the flat scoring schema.

Complex graphs

The last snippet of examples will be concerned with building more complex graphs. Although the previous examples are great displays of the basic functionality of the algorithm they don’t exhibit the expressive power of the approach to any degree. This section will demonstrate this flexibility through a series of consecutive alignments and merges against the same graph and index.

We start by building a graph from all the sequences from the first set of tests:

```
./build_index.sh --input-sequences=ACGTATTAC,ACGGATTAC,
ACGTTTAC,ACGTAATTAC --index=complex.index --error-margin=1
--png=complex1
```

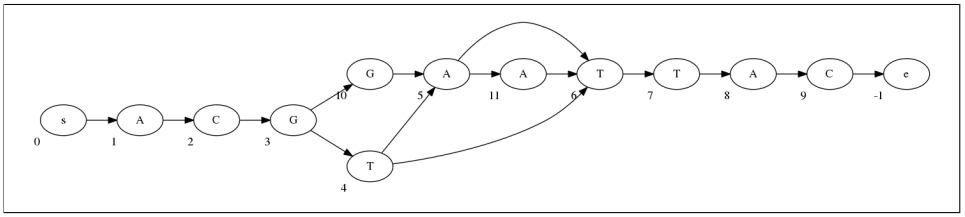


Figure 5.9: The reference graph made from the sequences “ACGTATTAC”, "ACGGATTAC", “ACGTTTAC”, “ACGTAATTAC”

- All sequences used in building the graph has a corresponding full path

We then merge in the sequences from the previous section:

```
./align_sequence.sh --index=complex.index
--align-sequence=ACGTGGTTAC --error-margin=2 --merge=true
./align_sequence.sh --index=complex.index
--align-sequence=ACGTACCTT --error-margin=4 --merge=true
--png=complex-merge
```

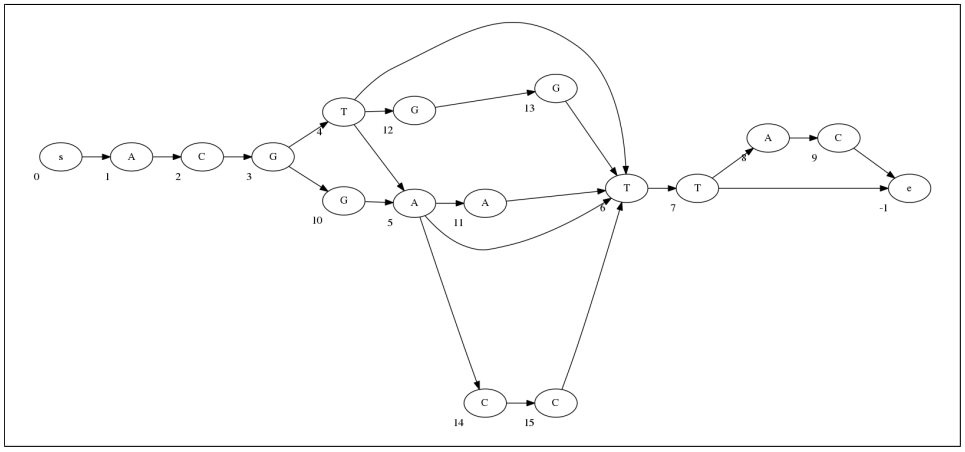


Figure 5.10: The two sequences “ACGTGGTTAC” and “ACGTACCTT” merged into the graph reference from figure 5.9

Although it is small, this is what we would consider to be a complex graph in the domain of genetic information. If we let branching factor denote the relationship between the number of vertices and the number of edges, the set of six sequences result in the branching factor $b \approx 1.4$. This indicates an average probability of variation of approximately 10% for every base, a level of variation which display the expressiveness of the model. Because the graph is a result of several levels of complex nesting it is even harder to provide intuitive statements as to what it should look like. We can still depend on the definitions to provide them for us. For instance we can again expect every input sequence to have a full path. We confirm this by aligning one of the sequences against the graph:

```
./align_sequence --index=complex.index
--align-sequence=ACGTGGTTAC --error-margin=0
--png=complex-align
```

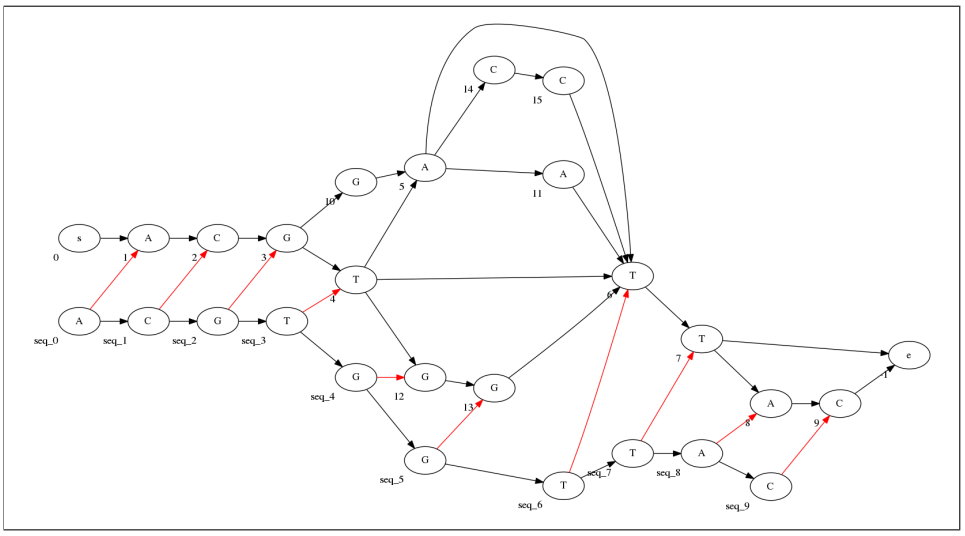


Figure 5.11: Aligning the sequence “ACGTGGTTAC” against the reference graph seen in figure 5.10

Because of the quick growth in complexity we let this be the last visualized result. We will in the next chapter move on to large datasets and automated validation when testing the efficiency of the approach.

Chapter 6

Performance testing

This chapter is concerned with testing the efficiency of the approach by running large scale tests on the implemented tool. There are two metrics which will be presented: The running time of the algorithm and the correctness of the achieved result. Most of the results are compared against running the same alignment with our own implementation of the PO-MSA algorithm. An explanation on how to find this implementation is given in Appendix C. We chose PO-MSA because of its intuitive nature, the easily deducible relationship between graph complexity and running time and, most importantly, because it is a non-heuristical approach guaranteeing a correct result every time.

6.1 Test data

These tests are meant to reflect usage in what would be an every day situation, and therefore use real genetic data. All of the sequences are FASTA-files from the MHC region fetched from the vg github repo [49] and the test-set provided by the sequence graphs tool [31]. The exact test sets are chosen to provide a variety of sequence lengths. In order to cover lengths where we found no sequences we created artificial sequences by cutting out suitable regions from longer sequences. All the sequences which are used can be found in the data-folder of the github repo of the tool.

Specifically, there are 8 main data-sets involved in the testing process:

- **mhc1.fa** A 700 bp long sequence from the MHC region (not specified more precisely where). Fetched from the docker repo of the sg project
- **primary.fasta** A 3345 bp long sequence from the HLA-A gene in the MHC region from the primary assembly of GRCh38. Originates from the vg github.
- **20k.fasta, 35k.fasta, 100k.fasta, 150k.fasta, 500k.fasta** Five subsequences of an alternate assembly of the MHC region of respectively 21.070bp, 35.770bp, 101.570bp, 144.480bp and 448.490bp. The

alternate assembly originates from the NCBI database[29] with id NT_167244.1.

- **mhc_full.fa** The previously mentioned full MHC assembly of 4.622.290bp.

Additionally, some of the tests use more specific data to test specific properties. This data will be presented before it is used.

In order to do alignments we need reads aswell as the data used in building the reference structure. These reads are generated by the read generator which is also described in Appendix C. The reads are generated by subtracting a sequence of a given length from the reference graph. In some cases we introduce noise to the reads proportional to a noise probability p. The noise is evenly divided among SNPs, insertions and deletions and is uniformly distributed over the entire length of the read. This is a model which is not meant to reflect sequencing errors[1], a simplification towards the most general version of the alignment problem.

In detail, the generation of a read r from a graph G happens as follows:

1. Choose a random vertex $v \in V \setminus \{s_G, t_G\}$ such that the smallest distance from v to t_g is larger than the chosen read size $|r|$
2. For $|r|$ steps:
   (a) Append $b(v_x)$ to the read r
   (b) Choose a random neighbouring vertex $v_y \in n_o(v_x)$ as the new v_x
3. When a read r has been generated, for $r_i \in r$
   (a) Choose a random floating point value $0 <= v <= 1$
      - If $v < (p/3)$ delete r_i
      - Else if $v < (2p/3)$ insert a random base $b \in \{A, C, G, T\}$ before r_i
      - Else if $v < p$ substitue r_i with a random base $b \in \{A, C, G, T\} \setminus \{r_i\}$
4. Output r

In order to provide reproducability the randomness in the reads are generated from a seed.

6.2 Validation

When an alignment is produced for a read we classify it as either as correct or not correct. Intuitively one could imagine correctness is determined

[1]Described in section 2.2.3

by whether the generated read aligns back to the path it was generated from. However, when noise is introduced an interesting phenomenon can occur: The modified read can become more similar to another path in the graph than its origin. This can also occur whenever there exists actual equal paths in the graph, typically in the case of repeats. In order to stick with mathemathical properties, our definition of optimality holds no relation to the origin of a read but is purely defined as the path which produces the highest possible alignment score. As PO-MSA is an exhaustive search we define optimally aligned as alignments which produce the same alignment score as the highest score found by PO-MSA. Consequently, as only the scores are compared, even when the approach produces a different alignment than PO-MSA, this is classified as optimal behaviour. Correctness is only discussed in the later stages of the chapter. When we do not add noise to the reads we are able to devises cases where the approach always provide correct results. In all tests where accuracy is omitted this is because the approach had a 100% success rate.

6.3 Time capturing mechanisms

For both the "Fuzzy context-based search" and the PO-MSA algorithm the time capturing mechanisms are built into the tool, using the Java System object. This allows us to wrap the time capturing of each individual constituent as close to the functional parts as possible in order to avoid unecessary overhead. When comparing tools the time was taken from the tool was started until the tool ended. Doing it this way has several disadvantages, which are discussed in Section 6.6. The time unit used throughout the chapter is milliseconds.

6.4 Building the index

The building of the index is the first step of the process realized through the build_index.sh script. To summarize this step consists of reading the input files, building the graph and generating the suffix trees. The build process was run 50 times on 6 different data sets, the averaged results can be seen in figure 6.1. The tool was not able to build an index for the largest input file with 4.5Mb because of insufficient memory[2]. Figure 6.2 shows the run time broken down into individual constituents.

[2]Run with 4Gb through the Java parameters -Xmx and -Xms

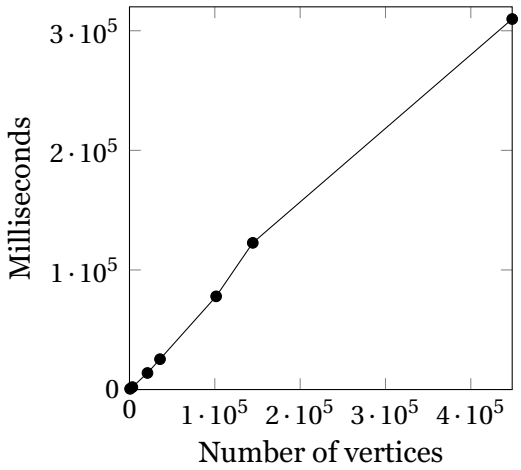
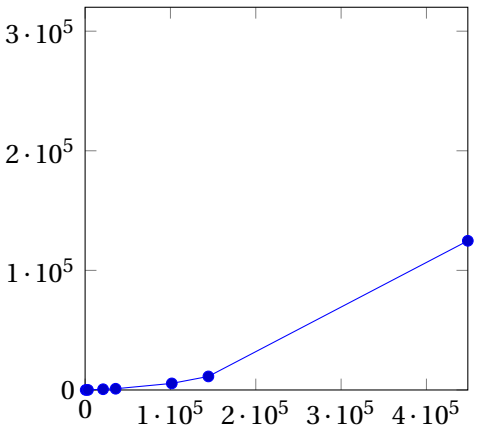
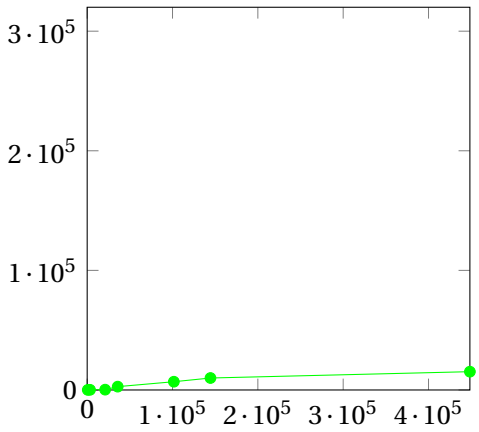


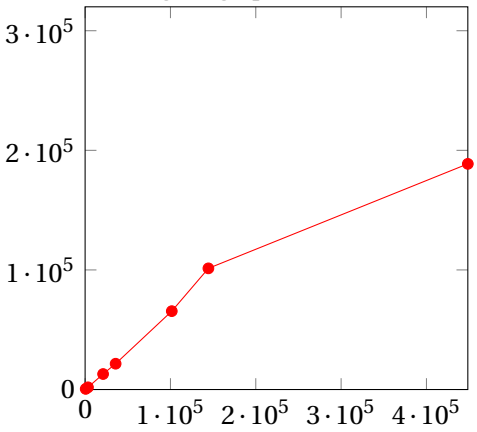
Figure 6.1: Runtime for the build index procedure as a function of the number of vertices



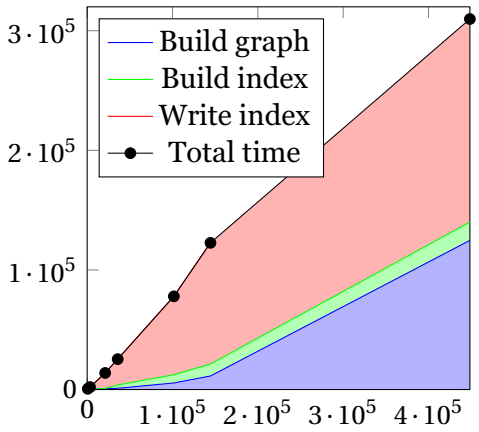
(a) Time used reading input files and building the graph



(b) Time used building the index



(c) Time used writing the index



(d) Total run time as a combination of the individual steps

Figure 6.2: Time used by the individual constituents of the build index process

6.5 Alignment

The alignment tests are run by the align_sequence.sh script, both with `--type=fuzzy` and `--type=po_msa` parameters. As a remainder to the reader, these are the variables which are in play:

- $|G|$ is the size of the graph
- λ is the allowed error margin
- $|s|$ is the length of the input sequence
- b is the branching factor of the graph
- p is the amount of noise added to the reads

As each of the subsequent sections are concerned with the impact of exactly one of these variables, the others are locked to a standard value:

- $|\mathbf{G}| = \mathbf{35.000}$
  Representing the mid-range of our test-sets.
- $\lambda = \mathbf{0}, \mathbf{p} = \mathbf{0.0}$
  We let alignment back to the origin represent the base case in the study.
- $|\mathbf{s}| = \mathbf{120}$
  Common read length for the Illumina HiSeq3000/4000 technology.
- $\mathbf{b} = \mathbf{1}$
  Calculations can be found in section 6.5.

Runtime as a function of graph size

We start by comparing the two alignment algorithms on different graph sizes. Figure 6.3 shows the average results over 50 runs on a log-log scale. Neither of the algorithms managed to handle the largest dataset of 4Mb because of insufficient memory. PO-MSA also had memory issues and were unable to produce results for the second largest dataset of 500kb. A linear scale plot of the 6 datasets which both algorithms managed to handle can be seen in figure 6.4

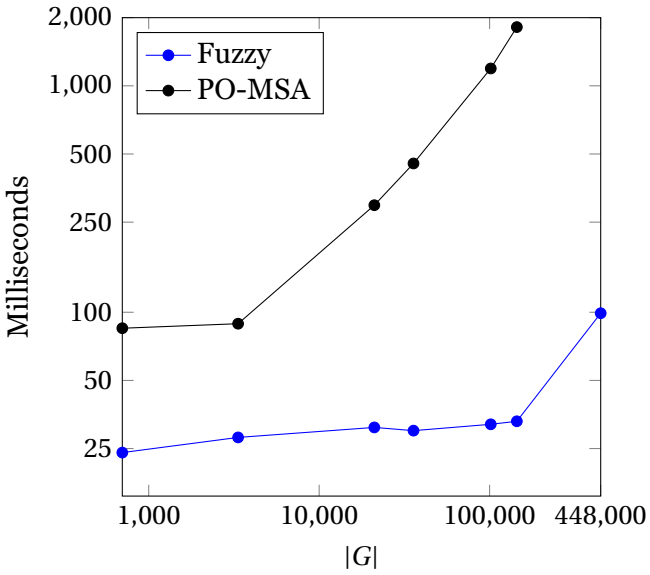


Figure 6.3: Runtime of the alignment process as a function of $|G|$ on a log-log scale for the 7 smallest datasets

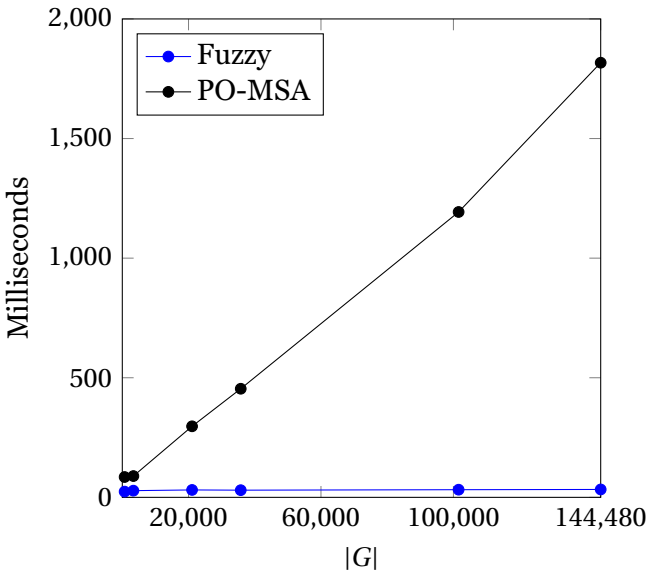


Figure 6.4: Runtime of the alignment process as a function of $|G|$ on a linear scale for the 6 smallest datasets

Runtime as a function of error margin

We vary the error margin by giving the algorithm different λ-values through the `--error-margin` parameter. The results from 50 runs at 5 different values are seen in figure 6.5. Figure 6.6 shows the same set of tests on the larger graph and includes a plot for an equivalent set of tests run with the `--parallellization=true` parameter.

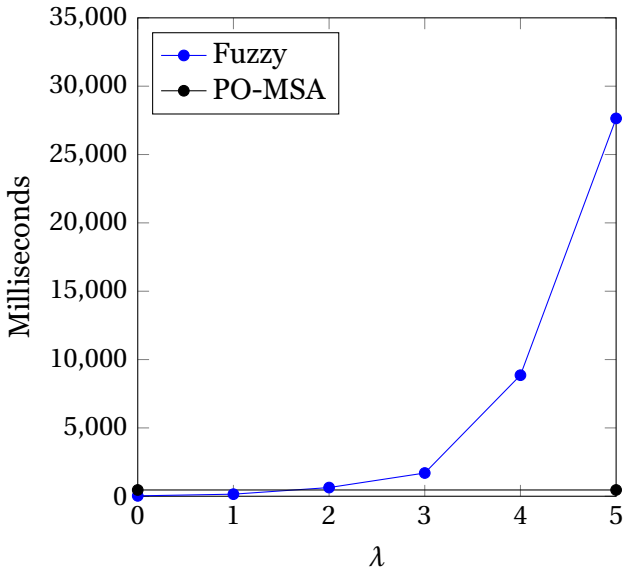


Figure 6.5: Runtime of the alignment process as a function of λ

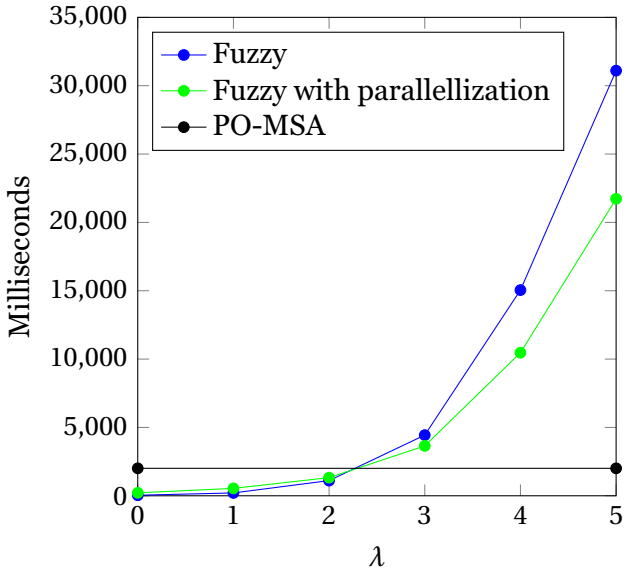


Figure 6.6: Runtime of the alignment process as a function of λ with $|G| = 150.000$ and `--parallellization=true`

Runtime as a function of sequence length

We vary the sequence lengths through changing the lengths produced by the read generator. The read lengths are chosen to reflect a variety of sequencing machines, displaying a diversity in read lengths [38][24]. Both the technologies, the lengths and the runtimes are listed in table 6.1. The number are visualized in figure 6.7

Technology	Read length	PO-MSA time	Fuzzy time
HiSeq2000 (min)	50	171	19
SOLiDv4	100	294	36
HiSeq3000/4000	120	545	39
Ion PGM	200	676	59
Sanger 3730xl (min)	400	1117	98
454 GS FLX	700	1600	167
Sanger 3730xl (max)	900	2121	194

Table 6.1: Running times for reads produced by different sequencing technologies for the PO-MSA and fuzzy search algorithm

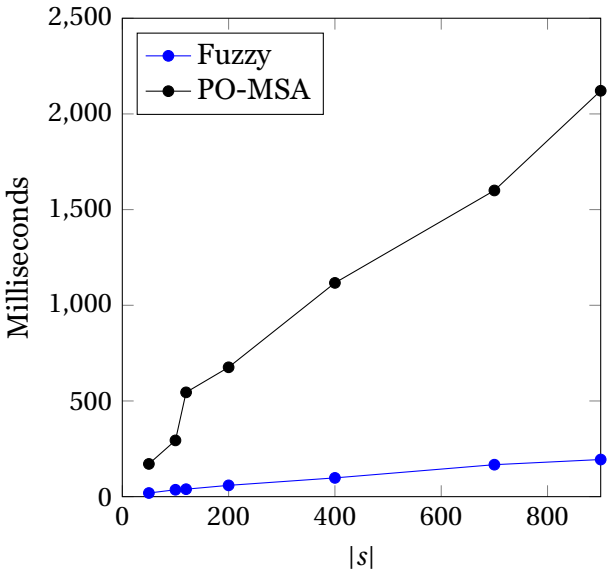


Figure 6.7: Runtime of the alignment process as a function of $|s|$

Runtime as a function of graph complexity

We let the branching probability b denote the complexity of our graph. This is a factor computed as a function of the relationship between the number of vertices and the number of edges. We created variation through generating a set of VCF files[3] with the read generator to provide a variety of values. The results of running alignments against each of the indexes can be seen in figure 6.8.

If we use the numbers from the 1000 genomes project from 2010 [9] we can approximate a branching factor $b \approx 1.0066$ for a population of 179 individuals [4]. At the completion of the project, ~88 million variants had been found in a population of 2504, indicating the factor $b \approx 1.029$ [8]. Both of these data points are visualized in the plot.

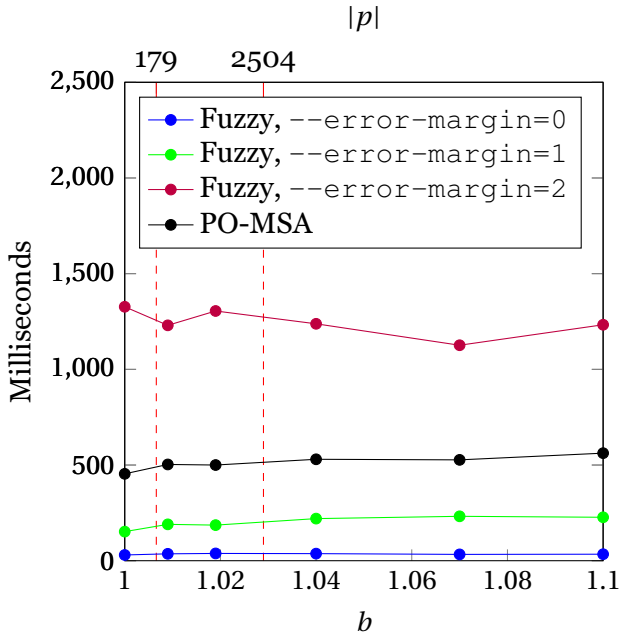
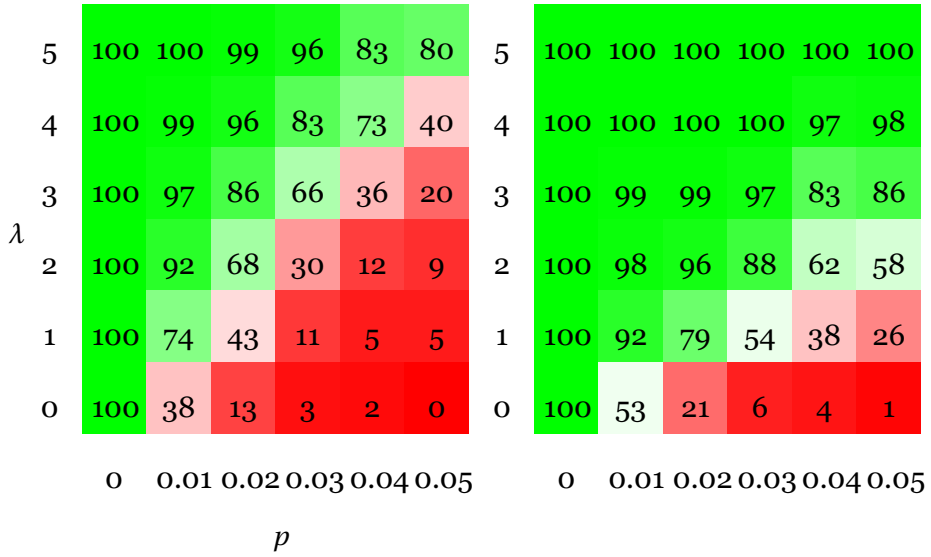


Figure 6.8: Runtime of the alignment process as a function of b. Datapoints depicting population sizes from earlier studies are shown as red vertical lines

[3]The variant file format mentioned in section 2.2.3
[4]14.894.361 SNPs + 3.019.909 indels + 21.075 structural variations / 2.420.000.000bp

Accuracy as a function of noise

In the previous tests we have guaranteed correct results through the tuning of input parameters. We will now introduce a level of uncertainty through introducing randomness to our reads through the noise parameter p. To some degree this is a futile exercise: We will get correct results when the number of modifications is lower than λ and empty alignments in the remaining cases, mirroring the distribution of the underlying randomness. This is however interesting as a depiction of a real life situation where the noise is to some degree uncertain. The percentage of correctly aligned reads over a total of 3600 runs with a variety of settings is seen in 6.9a. In this set of tests we also include the results from the heuristical algorithm, shown in 6.9b.



(a) Results from the tool with default settings

(b) Results from the tool run with the parameter`--heuristical=true`

Figure 6.9: Percentage of correctly aligned reads as a function of both p and λ. Each cell in the heatmap represents one setting

6.6 Comparison with the sequence graphs tool

In this section we will compare the GraphGenome tool with the sequence graphs tool (sg) created by Novak et al. as an implementation of the algorithm presented in the article "Canonical, Stable, General Mapping using Context Schemes". On the github page [32] the creators state:

> "Indeed, the tool to align sequences to the index is currently unfinished."

Additionally, we did not manage to run the tool on larger datasets. For these reasons the sg tool has not been used as a benchmark for the remaining tests, but only been granted this section.

The two tools were compared in building the index and doing an alignment, the results can be seen in respectively figure 6.10 and 6.11. Because of the limitations with regards to graph size, we have introduced several smaller sample fasta-files found in their test-folder as input data. We have divided the execution time of the GraphGenome tool into "functional" time and file I/O to indicate the level of overhead which is expected when doing comparisons of execution times for complete tools.

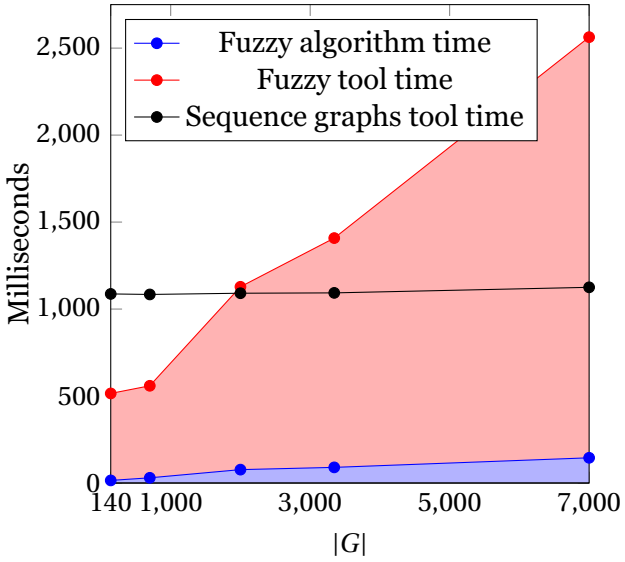


Figure 6.10: Time spent building the index by the two tools. The red line indicates the total time spent by the GraphGenome tool. The red section indicates file I/O, the blue section indicates time spent by the fuzzy search algorithm

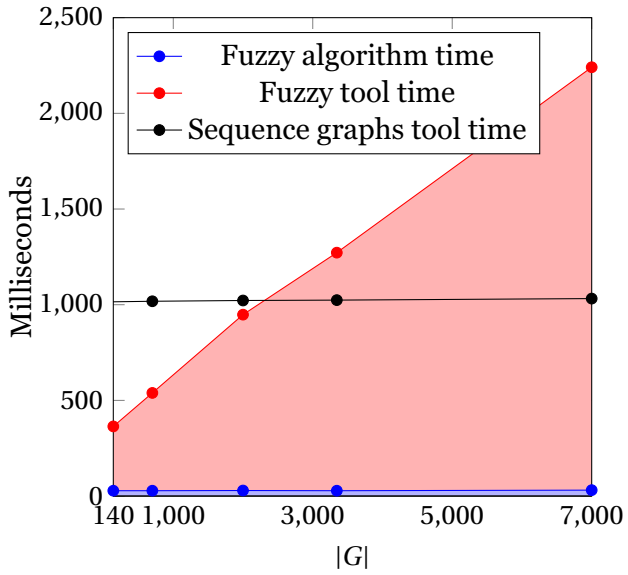


Figure 6.11: Runtimes of alignment by the two tools

The sg tool has a `--mismatch` parameter which works similarly to our `--error-margin` parameter by putting a bound on the allowed number of mismatches. The accuracy of the two tools over varying amounts of noise with a set mismatch and error-margin parameter can be seen in figure 6.12.

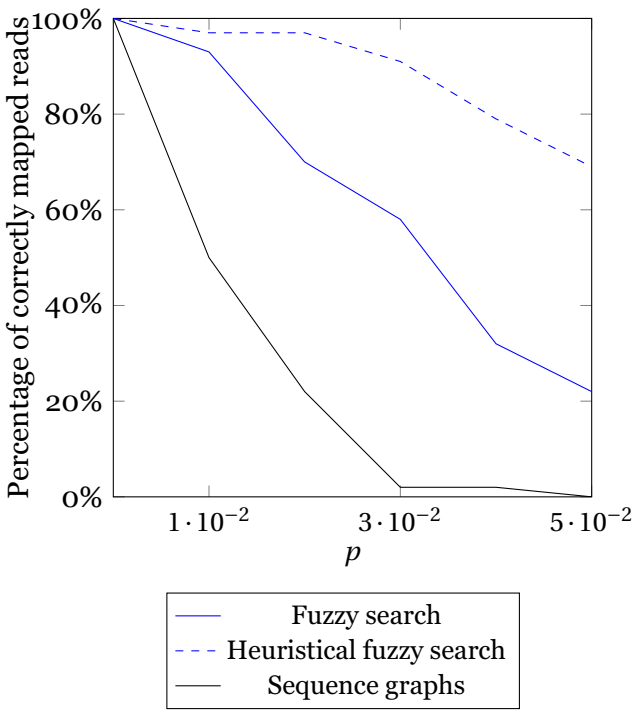


Figure 6.12: Accuracy of the two tools with $\lambda = 2$. The dashed line indicates the accuracy of the GraphGenome tool with the parameter `--heuristical=true`

Chapter 7

Discussion

We will in this chapter discuss the results which were presented in the preceding chapters. The chapter will highlight interesting outcomes of the tests and discuss the impact they have on the viability of the approach. The discussions will form the basis for the two subsequent chapters.

7.1 Is the approach correct?

We start by discussing the results from Chapter 5. Every test case in the chapter comes with a set of formal statements regarding the provided visualization, which represents the underlying result. We are not interested in arguing whether these statements are true or not: We want to discuss if the statements provide a valid basis for confirming the correctness of the approach.

In the introduction of the chapter, we describe omitting trivial statements to avoid tedious lists of uninteresting properties. By classifying a distinct set of traits as trivial, we implicitly classify another, disjoint set as non-trivial. We have not chosen statements from this set exhaustively: We have chosen a set of statements which we consider to be non-trivial, but still general enough to describe the properties of the most fundamental, underlying, mathematical version of the alignment problem. When we envision the approach being utilized for a specific biological problem, we would assume the need for a more specific set of statements. These could stem from domain knowledge from the exact biological question being answered, such as

> "Every valid alignment should only contain vertices from a separable subset of the population represented in the graph."

Or they could originate from a statistical analysis point of view, for instance

> "The fraction of input sequences traversing a path should impact the alignment score of every alignment against that path."

We claim these statements are unambiguously extensions to the set of statements provided in this thesis, and thus they strictly represent further specifications of the problem being solved. Although they might be necessary adjustments to turn the approach into an applicable solution to real life problems, we argue the algorithm as presented here has an innate value in its universality, as a proof of concept: A basis easily modifiable for more specific scenarios. We conclude that the approach is correct for the problem it is meant to solve, and although this problem is a simplified version it is situated at the core of more specialized applications.

The structural variation as it is presented in the chapter displays a weakness of the approach. In section 4.3 we mention the concept of a split alignment when discussing the heuristics, which can be a result of such variation. The approach is able to locate the data which is needed to identify a split alignment, but it is not able to handle it. We imagine this could be done through a more specified scoring schema, for instance one which allows a large gap. This represents an increase in conceptual complexity and is left for others.

7.2 Is the approach efficient?

We will move on to discuss the results found in Chapter 6. Specifically, there are three complexity related characteristics with these results we find interesting: The efficiency of the approach under optimal conditions, the complexity in relation to introducing fuzzyness and the indexation. The three subsequent sections will summarize these characteristics, discuss the nature of their results and the effects this has on the viability of the approach. The results from the accuracy tests will be discussed separately in section 7.4 and also to a large degree form the fundament for possible future work discussed in Chapter 9.

Optimal conditions

We define the base case in the experiments done as the problem of aligning an unaltered read back to its origin. We test this by putting constraints on the parameters concerned with fuzzyness and vary the remaining variables, typically related to the complexity of the input data. The results can be seen in figures 6.3, 6.4, 6.7 and 6.8. The first set of tests were concerned with determining the relationship between execution time and the number of vertices in the graph. As expected PO-MSA shows a clear linear relation over the entire sample it was able to run. The fuzzy search is only dependant on the graph size to determine the depth of the suffix tree which is searched, and this logarithmical relationship looks almost constant in comparison. In the tests examining different sequence lengths, we also see a much smaller growth in our approach then what can be seen in PO-MSA, although both appear to be linear. The reason for the different linear factors can be caused by the number of vertices which are traversed by the exhaustive

search. Every such vertex has to do a number of operations correlated to the sequence length, by reducing this number we impact the total number of operations necessary. The tests concerned with branching factor shows no distinct indications on which approach should be preferred.

The results from these three tests lead us to conclude that the fuzzy search algorithm outperforms PO-MSA by a large factor in what we have defined as the base cases: Aligning an unaltered read back to its origin. This in itself is not an extremely impressive result: It can be achieved through simpler solutions, for instance by using hashed k-mers of length $|r|$ from the graph as an index. What is interesting is that this is not the characteristic which was identified as the goal of the approach and then sought out in a vacuum: It emerges from a solution to a more general problem. This shows the approach is very tractable for the simplest use cases.

The exponential growth in relation to fuzzyness

We increase the complexity of the test cases by introducing fuzzyness through the error margin parameter λ. The results can be seen in figure 6.5, where a large exponential growth is displayed. This is in a way expected: Fuzzyness has the innate property of exponentially increasing the number of interesting possibilities. When we combine this with a dense search space of similar sequences, the growth rapidly increases. At some point, it is better to exhaustively let PO-MSA test the possibilities we actually have instead of letting the fuzzy search generate and search for variants. The border between the two is represented where the two functions cross in the figure.

The extreme growth and the early cross-over point between the two algorithms in the figure indicate that our approach has limited applications when faced with large amounts of variation. In figure 6.6 there are results which can refute this conclusion to some degree. We can see that a simple parallelization seems to put a linear bound on the exponential growth. There still exists room for large optimizations through parallelizing the suffix tree search[1]; we included a simple version to display the powerful effect. However, the most important point is seen in relation to graph size: Although the function grows fast, the starting point is decided by the "base case" of no fuzzyness, a behaviour we know will improve as the size of the input data increases. Figure 7.1 shows the results from the larger data set projected onto the plot displaying the results from the smaller data set[2]. The fact that these two factors do not seem to interact with each other results in an increase in tractability as the data sets grow: A very desirable trait when dealing with real genetic data.

Another important argument can be made as the graph size increases in comparison to the read length. There will exist a larger number of

[1] As presented in section 4.4
[2] The results seen in figures 6.5 and 6.6

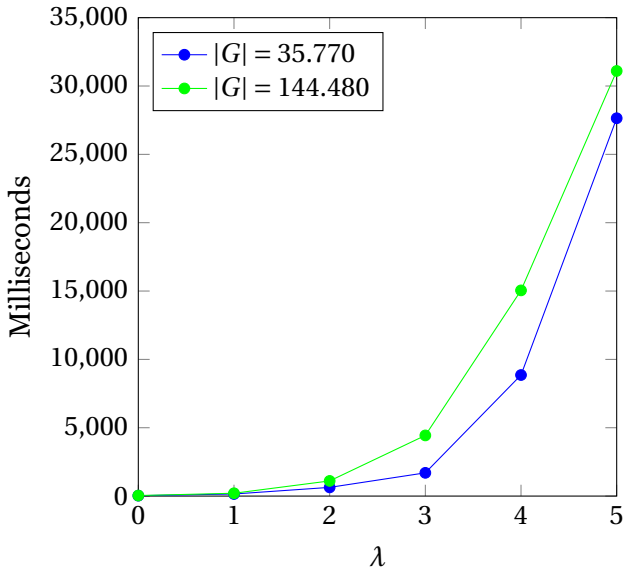


Figure 7.1: Runtime as a function of λ over different graph sizes

pure combinations of paths of length $|r|$ in the graph, which decreases the probability of not finding anything relevant. This, in turn, leads to a decreased need for a high λ value. This information leave us arguing that the approach still presents interesting opportunities when dealing with real data.

Indexation

One of the important results presented in the previous chapter is the amount of time used by the indexation process, as seen in figures 6.2d and 6.11. Most of this time is used by slow interactions with the file system, i.e. by reading and writing a large index. This thesis has not at all been concerned with the tractability of this process. Although the results seem severe there exists solutions both for better compression [3][53], better serialization [18] and smarter interactions with slower hardware [52, Section 4.7]. Ultimately one could argue that in a real life scenario the index should be kept in memory on a supercomputer [6][37]. Importantly, we are not concluding that it is fair to completely remove the indexation factor when examining the results. We are rather stating this is a key component of the developed approach, but the work is left for others.

7.3 A comparison between the sequence graphs tool and the "fuzzy context-based alignment" tool

In the previous chapter, we presented some results from a set of tests run as a comparison between the sg tool and our tool. We made some arguments

as to why this comparison was hard to do, and what problems we faced. Because of these, we will in this section focus on the conceptual differences in the two algorithms and the impact the divergence between them have on the results, both in light of the alignments they produce and what differences in time complexity can be expected. We will start by describing the approaches step by step to have a more fine-grained picture for the comparison. Throughout this section, we will refer to the approach developed by Novak et al. as sg and our approach as fuzzy search.

Both algorithms start out by searching for contexts. Already here there is a separation through the requirement for uniqueness, but we see this as a clear design choice taken based on domain knowledge. When we also take into account the previously described triviality of changing between the two choices we see this as a difference which is not that interesting in this setting. When the contexts have been found, the algorithms start to diverge. The fuzzy search picks out the vertice in "center" of the context and stores it in a candidate set. We later search through the candidate sets with an exhaustive search algorithm. Sg locks the entire context on to the string and seeks to increase the locked portion by combining overlapping contexts. The overlapped contexts are further expanded by combining them with allowed gaps in between, and finally doing a bounded search to fill in the remaining gaps.

We can go further in comparing the two by describing the sg algorithm in terms of the data structures used when presenting our approach. Because of the uniqueness restriction, each candidate set would either contain a single vertex or be empty. Both singular contexts and the combination of these would be represented by consecutive candidate sets containing consecutive vertices. The second search done to combine non-overlapping contexts would be similar to the search we do, a bounded search seeking to find "missing" candidate vertices for a short consecutive number of candidate sets. The final search is required when we have consecutive empty candidate sets spanning a number of indexes larger than a given mismatch parameter. This search could, for instance, be implemented as a PO-MSA search starting in the vertex in the last non-empty candidate set preceding the gap and ending in the vertex in the first non-empty candidate set succeeding the gap.

When we have both algorithms on this level of detail, the separation between them becomes apparent. Our approach is based on the assumption that *if we localize all interesting areas, an optimal solution is bound to be in the combination of them*. The sg approach seems based on an assumption along the lines of *if we find enough uniquely identifiable areas, we can combine these into a good solution*. We can see a clear cut distinction between a non-heuristical and a heuristical approach, which will have an effect on the quality of the results. There are two points which separate the algorithms in regards to runtime: Firstly, we search for a complete number of candidates, a number which grows exponentially

related to the allowed fuzzyness. We also do an exhaustive search on all the possible recombinations, another exponential factor. The results from this separation should become apparent in both algorithms when faced with non-optimal cases, represented by searching for strings which do not have a good counterpart in the graph. Sg would see a decrease in quality of the results while our approach sees a growth in runtime. Both can argue these are cases they are not meant specifically to handle.

7.4 Heuristical applications

The largest disadvantage of the approach which became apparent through the performance testing was the exponential growth in relation to introducing fuzzyness. Figure 6.9a depicts the accuracy of the algorithm as we modify the error margin parameter for datasets with varying amounts of noise. In figure 6.9b we can see the same set of tests run on the heuristical algorithm. The figures indicate that this modification can achieve better results with a lower error margin, a trait which will increase the tractability of the approach by limiting the factor which has the greatest impact on the time complexity. We will start this section by formally arguing that this observation is correct.

The heuristical functionality in the modified version of the algorithm will only influence the execution whenever optimality can not be guaranteed. This means the heuristical version will always find every optimal alignment which the non-heuristical algorithm can find. Because the two algorithms are equivalent in this regard, one might think the burden of the heuristicality would lie in introducing ambiguity. In our controlled test environment we have decided accuracy through comparing the results with a benchmark; A setting which is less viable in real life applications. The algorithm does, however, have built in a self-validating procedure: Whenever we find an alignment with a score $\varphi_A \geq T$ we can be certain it is optimal whether the algorithm is run heuristically or not.

In cases where the score drops below T we are faced with ambiguity we can not get around. We can still make assumptions on the quality of the alignment based on the input data: The heuristical algorithm will work better with input sequences which have a uniform distribution of noise. When the variation is spread out across multiple contexts, it will be able to identify candidate vertices with a lower λ. The number of correctly recognized candidate vertices which make it through the pruning has a direct impact on the final result. We can also assess the quality based on the difference in φ_A and T. These metrics provide us with tools for deciding when the heuristical result is good enough, and when we need to increase the error margin. The fact that the heuristical algorithm is always correct when the non-heuristical is and we have a method for validating it, shows it is a powerful addition to the regular approach.

We finished the last section by classifying the sequence graphs algorithm as heuristical and our approach as non-heuristical. We discussed the implications this has and argue they both provide valuable elements in the search for an optimal solution to the problem of alignment. If we let these two represent the extremes in a space we argue our heuristical modification is somewhere in between them. It is less complex than the normal fuzzy search algorithm due to the decreased requirement for a high error margin, but still more exhaustive than sg. It is important to once again point out that the progress towards sg in terms of reducing complexity does not come at the cost of a reduction in accuracy. This is an enticing trait for a heuristical modification.

We will present one more possibility for our approach in combination with other heuristical methods: A tool for validation. Running an alignment on a different heuristical tool produces a score, which can be used to calculate an error margin. Running our tool with the error margin results in one of three cases:

- An alignment with the same score, meaning one of these exist
- No alignment, meaning the score is faulty and the alignment is wrong
- An alignment with a higher score, meaning the heuristical method was wrong

The first case provides an obvious value if it is combined with a procedure for checking whether a given alignment has a given score. The second case will discover a bug in the heuristical procedure, which is definitively valuable, but not on a conceptual level. The third case is very interesting as it does not necessarily reveal a fault in the heuristical implementation, but more so a weakness in the heuristic used. When utilizing our approach as a validator, one can set an upper bound to the values of λ which should be validated to stay in the feasible region. This allows us to create a set of confirmed optimal, high scoring reads and a set of heuristically aligned, low scoring reads. This is information which is interesting for instance if the result is a set of variants.

Chapter 8

Conclusion

The aim of this thesis was to present the approach we developed, and through testing its validity and efficiency determine whether it represents a feasible solution to the problem of aligning text strings against graphs. We have shown that the implementation provides expected results and argued for the validity of these results as a solution to the most general form of the problem. We have also presented modifications we see as necessary for this approach to be translated towards more specific biological problems. The performance results were more ambiguous, revealing that the algorithm has both strengths and weaknesses. We have discussed both of these in the preceding chapter, while also proposing possible solutions to some of the arising problems. The fact that there are still shortcomings leads us to conclude that our approach can not serve as an optimal solution to the alignment problem. However, because of the feasibility shown by the heuristically modified algorithm, we do conclude it is an important step along the way towards this goal. We will briefly present our thoughts on where this path could lead in the subsequent chapter.

Chapter 9

Future work

In this chapter we will present our thoughts around what we assume to be necessary adjustments and expansions to the approach presented in this thesis to solve specific biological problems concerned with alignment problems where the reference genome is a graph. We will do this at various levels of abstraction, starting out with the explicit changes needed in the implementation to create a tool viable for answering actual biological questions. There are several details which would need to be introduced, such as the possibility of searching for reverse complementary strings and storing the origin of the input data. We consider these domain-specific, and trivial to implement. At this level of abstraction we also include the optimization to parallelization which is sketched out in section 4.4.

The rapid growth seen in the size of the index is a more conceptual problem to solve, but one we consider necessary to make the approach feasible for real life datasets. We displayed some possible solutions in section 7.2. A related problem is the one concerned with the exponential growth in time complexity with regards to the fuzzyness. This is a property which lies in the index, as it reflects the exponential increase in possibilities directly on the number of computations. Because the size of the index is already problematic, the naive solution of increasing its complexity is not satisfying. One possible solution to this problem is to decrease the number of searches. This can for instance be done by searching for larger structures, much like Paten et al. when they look up entire contexts instead of singular vertices. Our algorithm works as it does in order to preserve a level of flexibility until the exhaustive search. We believe this is a problem which can be solved by devising an algorithm for finding combinations of larger sequences, which will need to be sophisticated but can still see present an overall decrease in complexity.

We see the heuristical modification of the algorithm as the most interesting concept to continue pursuing. In section 7.4 we briefly sketch out systems for scoring the heuristical alignments. An actual implementation of this concept can go a long way to avoid having to deal with the complex problem instances which require high error margins. When these can be

avoided, the algorithm as we present it depicts a powerful solution.

Another interesting property which emerges from the heuristical version of the alignment is the possibility for aligning split sequences. We especially consider this interesting in the cases where the alignments are not split apart by a large gap, but when the split happens across distinct branches in the graph. These are the cases when a naive distance metric is no longer intricate enough, which is when the flexibility presented by graphs can really unfold.

Bibliography

[1] Göran Andersson et al. 'Retroelements in the human {MHC} class {II} region'. In: *Trends in Genetics* 14.3 (1998), pp. 109–114. ISSN: 0168-9525. DOI: http://dx.doi.org/10.1016/S0168-9525(97)01359-0. URL: http://www.sciencedirect.com/science/article/pii/S0168952597013590.

[2] Kenneth A. Berman and Jerome L. Paul. *Algoritms: Sequential, Parallell and distributed*. Thomson/Course Technology, 2005.

[3] M. Burrows and D. J. Wheeler. 'A block-sorting lossless data compression algorithm'. In: (1994). URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

[4] F. Chiaromonte, Yap VB and W. Miller. 'Scoring pairwise genomic sequence alignments'. In: (2002). URL: http://www.ncbi.nlm.nih.gov/pubmed/11928468.

[5] Deanna M. Church et al. 'Extending reference assembly models'. In: *Genome Biology* 16.13 (2015). URL: http://doi.org/10.1186/s13059-015-0587-3.

[6] P. C. Church et al. 'Design of multiple sequence alignment algorithms on parallel, distributed memory supercomputers'. In: *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Aug. 2011, pp. 924–927. DOI: 10.1109/IEMBS.2011.6090208.

[7] International Human Genome Sequencing Consortium. 'Initial sequencing and analysis of the human genome'. In: *Nature* 409 (6822). URL: http://dx.doi.org/10.1038/35057062.

[8] The 1000 Genomes Project Consortium. 'A global reference for human genetic variation'. In: *Nature* 526 (7571). DOI: http://dx.doi.org/10.1038/nature15393.

[9] The 1000 Genomes Project Consortium. 'A map of human genome variation from population-scale sequencing'. In: *Nature* 467 (2010). URL: http://dx.doi.org/10.1038/nature09534.

[10] The ENCODE Project Consortium. 'An integrated encyclopedia of DNA elements in the human genome'. In: *Nature* 489 (7414). DOI: http://dx.doi.org/10.1038/nature11247.

[11] Alexander Dilthey et al. ‘Improved genome inference in the MHC using a population reference graph’. In: *Nature Genetics* 47 (6 2015). URL: http://dx.doi.org/10.1038/ng.3257.

[12] Jennifer L. Freeman et al. ‘Copy number variation: New insights in genome diversity’. In: *Genome Research* 16 (2006). URL: http://genome.cshlp.org/content/16/8/949.long.

[13] *Graphviz*. URL: http://www.graphviz.org/.

[14] *GRC Home*. Genome Reference Consortium. URL: http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/.

[15] *GRCh38*. Genome Reference Consortium. URL: http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/.

[16] Roger Horton et al. ‘Variation analysis and gene annotation of eight MHC haplotypes: The MHC Haplotype Project’. In: *Immunogenetics* 60 (2008). URL: http://doi.org/10.1007/s00251-007-0262-2.

[17] Zamin Iqbal et al. ‘De novo assembly and genotyping of variants using colored de Bruijn graphs’. In: *Nature Genetics* 44 (2012). URL: http://dx.doi.org/10.1038/ng.1028.

[18] *JVM Serialization Benchmark*. URL: https://github.com/eishay/jvm-serializers/wiki.

[19] Birte Kehr et al. ‘Genome alignment with graph data structures: a comparison’. In: *BMC Bioinformatics* 15.1 (2014), pp. 1–20. ISSN: 1471-2105. DOI: 10.1186/1471-2105-15-99. URL: http://dx.doi.org/10.1186/1471-2105-15-99.

[20] Christopher Lee, Catherine Grasso and Mark F. Sharlow. ‘Multiple sequence alignment using partial order graphs’. In: *Bioinformatics* 18.3 (2002), pp. 452–464. DOI: 10.1093/bioinformatics/18.3.452. eprint: http://bioinformatics.oxfordjournals.org/content/18/3/452.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/18/3/452.abstract.

[21] Arthur M. Lesk. *Introduction to Bioinformatics*. Oxford University Press, 2014.

[22] Artur M. Lesk. *Introduction to genomics*. Oxford University Press, 2012.

[23] Heng Li and Richard Durbin. ‘Fast and accurate short read alignment with Burrows–Wheeler transform’. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760. DOI: 10.1093/bioinformatics/btp324. eprint: http://bioinformatics.oxfordjournals.org/content/25/14/1754.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/25/14/1754.abstract.

[24] Lin Liu et al. ‘Comparison of Next-Generation Sequencing Systems’. In: *Journal of Biomedicine and Biotechnology* 2012 (). DOI: 10.1155/2012/251364. URL: http://www.hindawi.com/journals/bmri/2012/251364/.

[25] Cristopher D. Manning, Prabhakar Raghavam and Hindrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] Shoshana Marcus, Hayan Lee and Michael C. Schatz. ‘SplitMEM: A graphical algorithm for pan-genome analysis with suffix skips’. In: *Bioinformatics* (2014). DOI: 10.1093/bioinformatics/btu756. eprint: http://bioinformatics.oxfordjournals.org/content/early/2014/11/13/bioinformatics.btu756.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/early/2014/11/13/bioinformatics.btu756.abstract.

[27] Paul Medvedev et al. ‘Error correction of high-throughput sequencing datasets with non-uniform coverage’. In: *Bioinformatics* 27.13 (2011), pp. i137–i141. DOI: 10.1093/bioinformatics/btr208. eprint: http://bioinformatics.oxfordjournals.org/content/27/13/i137.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/27/13/i137.abstract.

[28] Joong Chae Nal et al. ‘String Processing and Information Retrieval: 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings’. In: 2013. Chap. Suffix Array of Alignment: A Practical Index for Similar Data. URL: http://dx.doi.org/10.1007/978-3-319-02432-5_27.

[29] *National Center for Biotechnology Information*. National center for Biotechnology Information. URL: http://www.ncbi.nlm.nih.gov/.

[30] Ngan Nguyen et al. ‘Research in Computational Molecular Biology: 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, USA, April 2-5, 2014, Proceedings’. In: ed. by Roded Sharan. Cham: Springer International Publishing, 2014. Chap. Building a Pangenome Reference for a Population, pp. 207–221. ISBN: 978-3-319-05269-4. DOI: 10.1007/978-3-319-05269-4_17. URL: http://dx.doi.org/10.1007/978-3-319-05269-4_17.

[31] Adam Novak. *Sequence graphs*. URL: https://hub.docker.com/r/adamnovak/sequence-graphs/.

[32] Adam Novak. *Sequence graphs github*. URL: https://github.com/adamnovak/sequence-graphs.

[33] A. Novak et al. ‘Canonical, Stable, General Mapping using Context Schemes’. In: *ArXiv e-prints* (Jan. 2015). arXiv: 1501.04128 [q-bio.GN].

[34] B. Paten, A. Novak and D. Haussler. 'Mapping to a Reference Genome Structure'. In: *ArXiv e-prints* (Apr. 2014). arXiv: 1404.5010 [q-bio.GN].

[35] Benedict Paten et al. 'Cactus graphs for genome comparisons'. In: *Journal of Computational Biology* (2011). URL: http://online.liebertpub.com/doi/abs/10.1089/cmb.2010.0252.

[36] PA. Pevzner, H. Tang and MS. Waterman. 'An eulerian path approach to DNA fragment assembly'. In: *Proceedings of the National Academy of Sciences* 98 (2001).

[37] MJ Puckelwartz et al. 'Supercomputing for the parallelization of whole genome analysis'. In: *Bioinformatics* (2014). DOI: 10.1093/bioinformatics/btu071. eprint: http://bioinformatics.oxfordjournals.org/content/early/2014/02/12/bioinformatics.btu071.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/early/2014/02/12/bioinformatics.btu071.abstract.

[38] Michael A. Quail et al. 'A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers'. In: *BMC Genomics* 13.1 (2012), pp. 1–13. ISSN: 1471-2164. DOI: 10.1186/1471-2164-13-341. URL: http://dx.doi.org/10.1186/1471-2164-13-341.

[39] Melanie Schirmer et al. 'Insight into biases and sequencing errors for amplicon sequencing with the Illumina MiSeq platform'. In: *Nucleic Acids Research* (2015). DOI: 10.1093/nar/gku1341. eprint: http://nar.oxfordjournals.org/content/early/2015/01/13/nar.gku1341.full.pdf+html. URL: http://nar.oxfordjournals.org/content/early/2015/01/13/nar.gku1341.abstract.

[40] Korbinian Schneeberger et al. 'Simultaneous alignment of short reads against multiple genomes'. In: *Genome Biology* 10.9 (2009), pp. 1–12. ISSN: 1465-6906. DOI: 10.1186/gb-2009-10-9-r98. URL: http://dx.doi.org/10.1186/gb-2009-10-9-r98.

[41] Marcel H. Schulz et al. 'Fiona: a parallel and automatic strategy for read error correction'. In: *Bioinformatics* 30.17 (2014), pp. i356–i363. DOI: 10.1093/bioinformatics/btu440. eprint: http://bioinformatics.oxfordjournals.org/content/30/17/i356.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/30/17/i356.abstract.

[42] Peter H. Sellers. 'The theory and computation of evolutionary distances: Pattern recognition'. In: vol. 1. 1980. DOI: http://dx.doi.org/10.1016/0196-6774(80)90016-4.

[43] Michael Sipser. *Introduction to the Theory of Computation*. CENGAGE Learning, 2013.

[44] Simone Sommer. 'The importance of immune gene variability (MHC) in evolutionary ecology and conservation'. In: *Frontiers in Zoology* (2005). URL: http://doi.org/10.1186/1742-9994-2-16.

[45] *The JUnit Framework*. JUnit. URL: http://junit.org/junit4/.

[46] *The Serialization interface: Java API documentation*. Java. URL: https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

[47] E. Ukkonen. 'On-line construction of suffix trees'. In: *Algorithmica* 14.3 (), pp. 249–260. ISSN: 1432-0541. DOI: 10.1007/BF01206331. URL: http://dx.doi.org/10.1007/BF01206331.

[48] *Understanding the birthday paradox*. BetterExplained. URL: http://betterexplained.com/articles/understanding-the-birthday-paradox.

[49] *Variation graphs*. vgteam. URL: https://github.com/vgteam/vg.

[50] Xin Victoria Wang et al. 'Estimation of sequencing error rates in short reads'. In: *BMC Bioinformatics* 13.1 (2012), pp. 1–12. ISSN: 1471-2105. DOI: 10.1186/1471-2105-13-185. URL: http://dx.doi.org/10.1186/1471-2105-13-185.

[51] Ziqi Wang et al. 'A Fast and Accurate Method for Approximate String Search'. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*. HLT '11. Portland, Oregon: Association for Computational Linguistics, 2011, pp. 52–61. ISBN: 978-1-932432-87-9. URL: http://dl.acm.org/citation.cfm?id=2002472.2002480.

[52] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Pearson Education, 2007.

[53] J. Ziv and A. Lempel. 'A universal algorithm for sequential data compression'. In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

Appendices

Appendix A

Proving optimality

Because the description of the approach is rather comprehensive, we chose to include a more brief proof that the approach is correct. Because the second step of the algorithm is exhaustive we only need to prove the first step: Every vertex part of an optimal alignment is passed on from the suffix tree search.

We have a graph G and a string s such that there exists an optimal alignment A with a score $\$_A$. We have an error-margin λ such that the maximal achievable alignment score $max(s)$ is smaller than $\$_A + \lambda$. Because we are using consistent scoring schemas we know $max(s) = align(s, s)$. We also know $max(s + c) = max(s) + mappingScore(c, c)$ for any $c \in \Sigma$. Thus we can view $max(s)$ as a discrete function of the length of s, where the increase between any two indices are maximal for s.

We now assume the opposite case of what we are trying to prove:

Assumption 1
There exists a vertex v which is part of an optimal alignment with a score $\varphi_A > max(s) - \lambda$ which is pruned away from in the suffix tree search.

Because v is pruned away it is not a part of any contexts which has a context score higher than $T_c = max(s') - \lambda$ for any substring $s' \in s$. If we pick out the highest scoring context c this has a context score $T_c < max(s') - \lambda$. There are two cases to cover: Either $s' = s$ or $|s'| < |s|$. In the first case we can let the context score T_c also denote the alignment score φ_A. We insert $s' = s$ into the calculation and find $\varphi_A < max(s) - \lambda$ which contradicts the assumption. In the second case of $|s'| < |s|$ there exists a part of the string s which is not covered by the substring s' aligned against the context. We call this part $\bar{s}$. We let x denote the value which is lost when aligning c against s' such that $T_c = max(s') - \lambda - x, x > 0$. In order for the alignment of the full string $s' + \bar{s}$ to achieve a score $\varphi_A > max(s) - \lambda$, $\bar{s}$ has to be aligned against contexts such that the sum of their scores $T_c = max(\bar{s}) + x, x > 0$. From the previous definition of max we know that no alignment of any of the substrings $\bar{s} \in s$ can achieve a score higher than $max(\bar{s})$, which means this is impossible. Thus the max score for any alignment containing this

context is $\varphi_A = max(s) - \lambda - x, x > 0$ which also contradicts the assumption. Because both the exahustive cases lead to contradictions, the assumption is invalid.

Appendix B

Average case complexity analysis

We let G be a reference graph and s and input sequence. λ denotes the allowed error margin under the negated edit distance scoring schema.

For every candidate vertice v_x in every candidate set V'_i, $0 <= i < |s|$, we need to find the distance to every preceding vertice in every preceding candidate set V'_{i-j} for $1 <= j <= \lambda + 1$. We let this number be denoted by $dist(G, s)$

$$dist(G, s) = \Sigma_{i=0}^{(|s|-1)} |V'_i| \Sigma_{j=1}^{(\lambda+1)} |V'_{i-j} \tag{B.1}$$

If we assume the contexts in G and the substrings in s are both normally distributed over the space of all possible contexts, the size of any two candidate sets are interchangable. This is not an entirely true assumption in the cases of shorter contexts at the beginning and end of s, but the impact fo these fades as $|s|$ grows compared to $|c|$. We let $avg(V'_x)$ denote the average size of the candidate sets. We can use this approximation to contract the previous equation

$$\begin{aligned} dist(G, s) &= \Sigma_{i=0}^{(|s|-1)} |V'_i| \Sigma_{j=1}^{(\lambda+1)} |V'_{i-j} \\ &= (|s| - 1) avg(V'_x)(\lambda + 1) avg(V'_x) \\ &= (|s| - 1)(\lambda + 1) avg(V'_x)^2 \end{aligned} \tag{B.2}$$

The sizes of these sets are given by the number of valid contexts, $num(c, \lambda)$, multiplied by the probability that a context exists, $prob(c)$. If we set $\lambda = 0$ we allow no fuzzyness and are thus looking for a single context

$$num(c, 0) = 1 \tag{B.3}$$

When increasing λ by one we are allowing the search to branch out exactly once from the existing branches. There are 3 possibilities for branching out in the suffix tree at each of the $|c|$ levels, one possibility for each other character. Each of the branches adds another legal context to the search

$$num(c, \lambda) = 1 + 3|c| * num(c, \lambda - 1) \tag{B.4}$$

If we overestimate the number of possible branches by letting the search include every branch which has already been searched this number can be simplified

$$num(c, \lambda) = 1 + 3|c|^{\lambda} \tag{B.5}$$

The probability of a context existing in the suffix tree is given by the number of actual contexts divided by the number of possible contexts. We approximate the number of actual contexts by the branching factor of the graph b raised to $|c|$ for every vertice. If we assume the branching factor is close to 1 this can be contracted to $|G|$

$$prob(c) = \frac{|G|}{4^{|c|}} \tag{B.6}$$

which means

$$avg(V'_x) = (1 + 3|c|^{\lambda}) * \frac{|G|}{4^{|c|}} \tag{B.7}$$

and

$$dist(G, s) = (|s| - 1)(\lambda + 1)((1 + 3|c|^{\lambda}) * \frac{|G|}{4^{|c|}})^2 \tag{B.8}$$

The distance search itself consists of doing a simple comparison in every vertice which is reachable by the searching algorithm. If we again assume $b \approx 1$ and we let λ put an upper bound on the length of the paths which are searched the searching algorithm visits exactly λ vertices. The final complexity for the entire operation is found by multiplying the complexity of the distance search with the times it is executed

$$O(search(G', s, \lambda)) = \lambda((|s| - 1)(\lambda + 1)((1 + 3|c|^{\lambda}) * \frac{|G|}{4^{|c|}})^2) \tag{B.9}$$

which can be shortened to

$$O(search(G', s, \lambda)) = \frac{\lambda^2 |s|(|c|^{\lambda}|G|)^2}{4^{|c|^2}} \tag{B.10}$$

Appendix C

The GraphGenome tool

The GraphGenome tool can be found on github at `https://github.com/estenpro/graph_genome_java`. Instructions for retrieval and usage of the tool can be found in the github readme.

For non-technical users a crude demo can be found at `http://graph-genome-demo.appspot.com/`.

The read generator can be found in the github repo in the file `src/main/java/ReadGenerator.java`. The PO-MSA implementation can similarily be found in `src/main/java/utils/AlignmentUtils.java`

Appendix D

The "birthday problem" and context lengths

The "birthday problem" is the problem of deciding the probability $B(n)$ that two people in a group of n share a birthday. We can let $B'(n)$ represent the opposing probability, that there is noone amongst the n which share birthdays. As the two are both exhaustive and exclusive we know that $B(n) + B'(n) = 1$ and thus that $B(n) = 1 - B'(n)$.

Deciding the probability that noone is sharing a birthday can be broken down to the individual people making up the selection. When we add a person to the selection we multiply the probability for the previous selection with the probability that the given person does not share a birthday, which is given by the probability of selecting an available day from the total number of days, 365. As we already know we are not interested in selections where two people share birthdays, the available days are equal to 365 minus the number of people in the selection, $n - 1$. The total result for a group of n people is thus given by the formula:

$$B'(n) = \frac{365-0}{365} * \frac{365-1}{365} * \frac{365-2}{365} * \ldots * \frac{365-(n-2)}{365} * \frac{365-(n-1)}{365} \tag{D.1}$$

which can be approximated using taylor series [48]:

$$B'(n) = e^{-(n^2/(2*365))} \tag{D.2}$$

which gives:

$$B(n) = 1 - e^{-(n^2/(2*365))} \tag{D.3}$$

Conveniently, we can apply the approximations used in the birthday problem to determine the probability of contexts being shared by two or more vertices. We let the number of actual contexts x replace the number of people, and the number of possible contexts y replace the days of the year. y is easily calculated by $4^{|c|}$, which is the number of all possible strings over the alphabet $\{A, C, G, T\}$.

$$y = 4^{|c|} \tag{D.4}$$

Computing x requires more work, so we chose to do another approximation to avoid complex calculations. We let b be the branching factor, such

that every vertice has approximately b neighbours. Every one of these neighbours has approximately b neighbours again, and so on for every one of the $|c|$ vertices which make up a context. This leads $b^{|c|}$ contexts for each vertice and $|G|b^{|c|}$ total contexts. If we assume our graphs are mostly linear we can set $b = 1$ to end up with a total of $|G|$ contexts.

$$x = |G| \tag{D.5}$$

Plugging these values into the previous formula we search for the smallest context length $|c|$ which gives us less than 1% probability of shared contexts

$$|c| = \min_{c}(1 - e^{-|G|^2/2*4^c} < 0.01) \tag{D.6}$$

These approximations might seem crude, but as previously stated the length of the contexts does not affect the correctness of the final result and thus we are only trimming efficiency. The functions made by varying $|c|$ in the approximation seem to do a jump from values exponentially decreasing from 1 to values logaritmically approaching 0 (Visualized in figure D.1). The values close to 1 represent contexts with a low probability of overlap, which gives deep suffix trees. Values close to 0 represent contexts with a lot of overlap, which gives high complexity in the search part done by the algorithm. A clear goal is to be a part of the "jumping section" as laying to close to either of the boundaries run a danger of overcompensating for the opposite effect. The somewhat arbitrarily chosen value 1% was decided as a means to avoid bloated indexes at the cost of a small runtime increase.

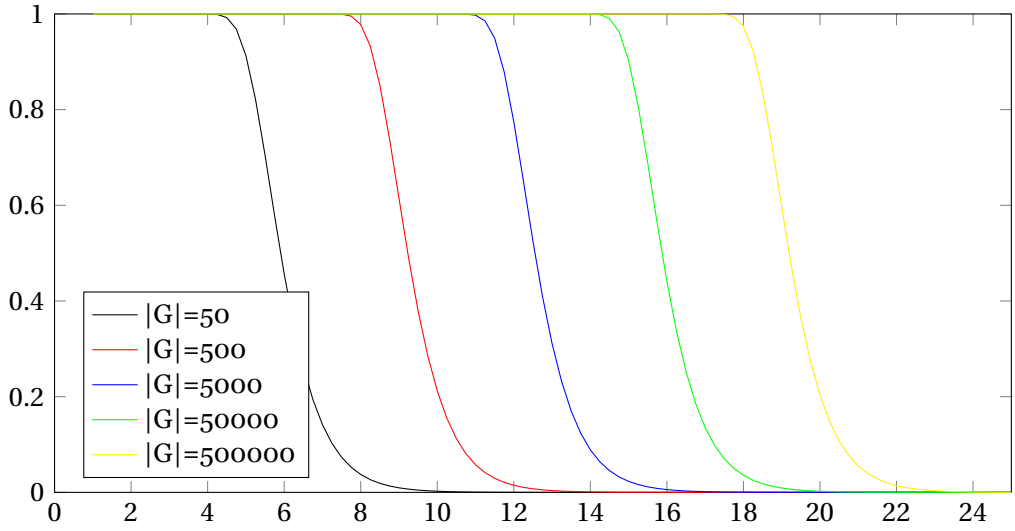


Figure D.1: The functions $y = B(|G|)$ provided by varying $x = |c|$ for different graph sizes