

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

26th April 2016

Abstract

TODO

Acknowledgements

Contents

1	Introduction	1
2	Background	3
2.1	Genetics	3
2.1.1	The central dogma	4
2.1.2	Variation	4
2.2	Genetic data, sequencing and string algorithms	5
2.2.1	Reference genomes	5
2.2.2	The human genome	5
2.2.3	Sequencing	6
2.2.4	Alignment	6
2.2.5	Dynamic programming	8
2.2.6	Suffix trees	9
2.2.7	Compression	9
2.3	Graph based genome representations	11
2.3.1	Model	11
2.3.2	Mapping	14
2.3.3	Alignment	15
3	Aims	17
4	The algorithm “Fuzzy context-based search”	19
4.1	The graphs	19
4.2	The alignment problem	22
4.3	“Fuzzy context-based search”	25
4.3.1	Constructing the candidate graph	25
4.3.2	Searching the newly formed graph	27
4.4	Heuristical applications	29
5	Implementation	31
5.1	Aligning sequences	31
5.1.1	Building the index	32
5.1.2	Generating the candidate graph	34
5.1.3	Searching the new graph with a modified PO-MSA search	36
5.1.4	Handling invalid threshold values	39
5.2	Necessary adjustments for a heuristical search	39
5.3	Parallelization	40

5.4	Merging aligned sequences	41
6	Validation of the approach	43
6.1	Test data	43
6.2	Tests	44
7	Performance testing	55
7.1	Test data	55
7.2	Validation	56
7.3	Time capturing mechanisms	57
7.4	Building the index	57
7.5	Alignment	59
7.6	Comparison with the sequence graphs tool	65
8	Discussion	67
8.1	Is the approach correct?	67
8.2	Is the approach efficient?	68
8.3	A comparison between the sequence graphs tool and the "fuzzy context-based alignment" tool	69
8.4	Heuristical applications	71
8.5	Conclusion	71
9	Future work	73
Appendices		79
A	Proving optimality	81
B	Average case complexity analysis	83
C	The GraphGenome tool	85
D	The "birthday problem" and context lengths	87

List of Figures

2.1	Examples of aligned text strings	7
2.2	The suffix tree (a) and suffix trie (b) of the string "bananas" .	10
2.3	Two proposed graph models displaying flexibility (a) and rigidity (b)	11
2.4	Various graph representations of the strings "ACGTTACG", "ACGTGTCA" and "ACGTTACG"	13
4.1	An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA"	21
4.2	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	23
5.1	A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown	34
5.2	The left suffix tree corresponding to the graph in 5.1	34
5.3	The resulting candidate sets for mapping the string "ATA" against the reference genome from fig. 5.1 with varying T values	36
5.4	Different scoring thresholds T yields different reference graphs	41
6.1	The reference graph made from the sequence "ACGTATTAC"	44
6.2	The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 6.1 . . .	45
6.3	The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 6.1 . . .	46
6.4	The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference seen in figure 6.1 . . .	47
6.5	The result of aligning (a) and merging (b) the sequence "ACGTTTAC" against the reference seen in figure 6.1 . . .	48
6.6	The result of aligning (a) and merging (b) the sequence "ACGTAATTAC" against the reference seen in figure 6.1 . .	49
6.7	The result of merging the sequence "ACGTGGTTAC" with the reference seen in figure 6.1	50

6.8	The result of merging the sequence “ACGTACCTT” with the reference seen in figure 6.1	51
6.9	The reference graph made from the sequences “ACGTAT-TAC”, “ACGGATTAC”, “ACGTTTAC”, “ACGTAATTAC” . .	51
6.10	The two sequences “ACGTGGTTAC” and “ACGTACCTT” merged into the graph reference from figure 6.9	52
6.11	Aligning the sequence “ACGTGGTTAC” against the reference graph seen in figure 6.10	53
7.1	Runtime for the build index procedure	58
7.2	Time used by the indexation process as a function of the number of vertices	58
7.3	Runtime of the alignment process as a function of $ G $	60
7.4	Runtime of the alignment process as a function of λ	61
7.5	Runtime of the alignment process as a function of λ with $ G = 150.000$ and --parallelization=true	61
7.6	Runtime of the alignment process as a function of $ s $	62
7.7	Runtime of the alignment process as a function of b	63
7.8	Percentage of correctly aligned reads as a function of both p and λ varies	64
7.9	Time spent building the index by the different tools	65
7.10	Runtimes of the two tools	66
7.11	Correctly mapped reads from the two tools with $\lambda = 2$	66
D.1	The functions $y = B(G)$ provided by varying $x = c $ for different graph sizes	88

List of Tables

2.1	The HOXD70 substitution matrix	8
2.2	The 2-dimensional array used for solving the edit distance problem for the strings S=“algorithm” and P=“logarithm” (Note: This follows regular ED scoring where every operation is penalized +1)	8
5.1	The 4 arrays used by the searching algorithm when using the candidate sets from Fig 5.3 and $T = -1$	37
7.1	Running times for different read lengths for the PO-MSA and fuzzy algorithms	62

List of Theorems

1	Definition (Graph based reference genome (Graph))	20
2	Definition (Graph genome vertex (Vertex))	20
3	Definition (Graph genome edge (Edge))	20
4	Definition (Graph genome path (Path))	20
5	Definition (Full path)	20
6	Definition (Incomplete path)	21
7	Definition (Path score)	21
8	Definition (Input sequence)	22
9	Definition (Alignment)	22
10	Definition (Maximal unmapped subsequence)	23
11	Definition (Alignment score)	23
12	Definition (The optimal alignment score problem)	23
13	Definition (The bounded optimal alignment score problem)	24
14	Definition (Graph genome weighted edge (Weighted edge))	27

Chapter 1

Introduction

TODO

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[FIND SOURCES]. We then step into the realm of technology and bioinformatics presenting techniques and concepts necessary in our own 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 still 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 has T and C has G, with which it can bind to form a *base pair* (bp). Larger numbers of base pairs are typically numbered with standard SI units (kb, mb, gb). 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 which strands are connected in a double helix structure. The two strands will have opposing directions, and every base in one of the strands will be connected to its complement. Because either of the strands are easily deduced from the other, DNA is usually represented by only one of them. We can then view DNA as a linear sequence of discrete units and represent it by text strings containing the four leading letters representing the nucleotides. The text strings representations often also contain the letter N, referencing *aNy base*. The genetic sequence of an individual is called the *genotype*. Observable traits of the individual 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 create 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*[16, 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 is coding regions[16, Chapter 4], the rest used to be referred to as *junk DNA*. We now know that the non-coding regions also holds important information.[REF]

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 [7], 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.[referanser](#)

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 are 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 of the most important concepts in handling this data and the field of bioinformatics. We will also present the MHC region, a genetic region which we used when 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* for a 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 genomes¹. The structure of a reference genome can increase computational tractability compared to storing a set of individual genomes, by reducing the double-storage of equal 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 [16]. The human reference genome is developed and maintained by the *Genome Reference Consortium*[9], the current version is called the GRCh38[10]. GRCh38 contains 261 alternate loci, spread over 178 out of 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 [5].

¹Covered in section 2.2.3

Major Histocompatibility Complex

The *Major Histocompatibility Complex* (MHC) is a genetic region spanning approximately 4.5-5 million base pairs (mb)[6][26]. In humans it is located on chromosome 6 and contains about 200 genes. MHC is a region known to contain genes which affect the functionality of the immune system [34]. Even more so MHC is known to be a highly variable region, containing variants that are directly associated with disease [11]. The high variability creates difficulties when comparing DNA sequences to determine genetic causes for the observed disorders.

2.2.3 Sequencing

During *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[30] which are combined into longer sequences through a process called *assembly*. When the sequenced individual belongs to a specie with a reference genome, reads are typically mapped to positions in the reference to determine their underlying order in what is called *mapping assembly*. In the opposing case overlap techniques[29] or de Bruijn graphs² are often used in what is known as *de novo assembly*[16, Chapter 1, p. 19].

The different sequencing technologies have varying degrees of errors introduced in their reads, often closely related to the sequencing cost[30]. The errors can take the form of both point mutations and larger structural variations. Reads produced by sequencing machines are typically prone to contain more errors in their peripherals. There exists efficient strategies for both estimating error rates [40] and correcting the reads[21] Can probably provide more citations.

2.2.4 Alignment

Sequence alignment is the process of determining 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) providing a *mapping score*. Throughout this thesis we will let the notation $\text{mappingScore}(c_1, c_2)$ denote the score for mapping two characters c_1, c_2 against each other. Alignment of DNA strings has several 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 one element in one of the string does not have a counterpart in the opposing string (Figure 2.1). When a gap occurs the resulting alignment is penalized according to the length of the gap, by a *gap penalty*. We will similar-

²Presented in 2.3.1

ily let the notation $gapPenalty(distance)$ denote the gap 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 in 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 around simple match/mismatch scores, which corresponds to the mathematical *Edit distance problem*³, or more complex scores (Table 2.1). These 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. There exists two main variants of the 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*.

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 arised 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 does 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. Both of the preceding techniques typically utilize heuristical methods in order to decrease the computational complexity.

Figure 2.1: Examples of aligned text strings

³Presented in detail in the next 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

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 represent the bottom of the recursion and is a value which can easily be computed without further lookups. 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 computational complexity while giving a guarantee for optimal results [1, 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 2-dimensional array to store the computed values (Figure 2.2). For two strings S and P , every index $[i, j]$ in the edit distance table represents the problem instance of the substrings $S[0 : i], P[0 : j]$.

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

Table 2.2: The 2-dimensional array used for solving the edit distance problem for the strings $S=$ “algorithm” and $P=$ “logarithm” (Note: This follows regular ED scoring where every operation is penalized +1)

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:

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

where $\text{score}(x, y)$ is an equality function returning 0 if the two elements are equal. The score for the entire problem instance 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 2-dimensional array and recursing upwards to the right. 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 where the vertices along path from the root to the vertex contains the characters of that suffix. Consequently, every substring has a path starting in the root node. A *compressed suffix trie*, or *suffix tree*, is a suffix trie in which every linear path is compressed into a single vertex (Figure 2.2b). Both suffix tries and suffix trees can easily be extended to hold collections of strings[1, Chapter 20]. A simple 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 [1, Section 20.6.1]. The tree can be constructed in linear time[37].

2.2.7 Compression

One major challenge when dealing with large amounts of genetic data is storage. In 1994, M. Burrows and D.J. Wheeler presented the "Burrows-Wheeler Transform" as an efficient compression scheme for text strings[2].

The algorithm rearranges the string by grouping together equal elements, a trait which is favorable for compression algorithms. The operation is reversible, meaning the original string can be easily computed from the transformed one. The approach has later been used for developing efficient, accurate alignment algorithms[17].

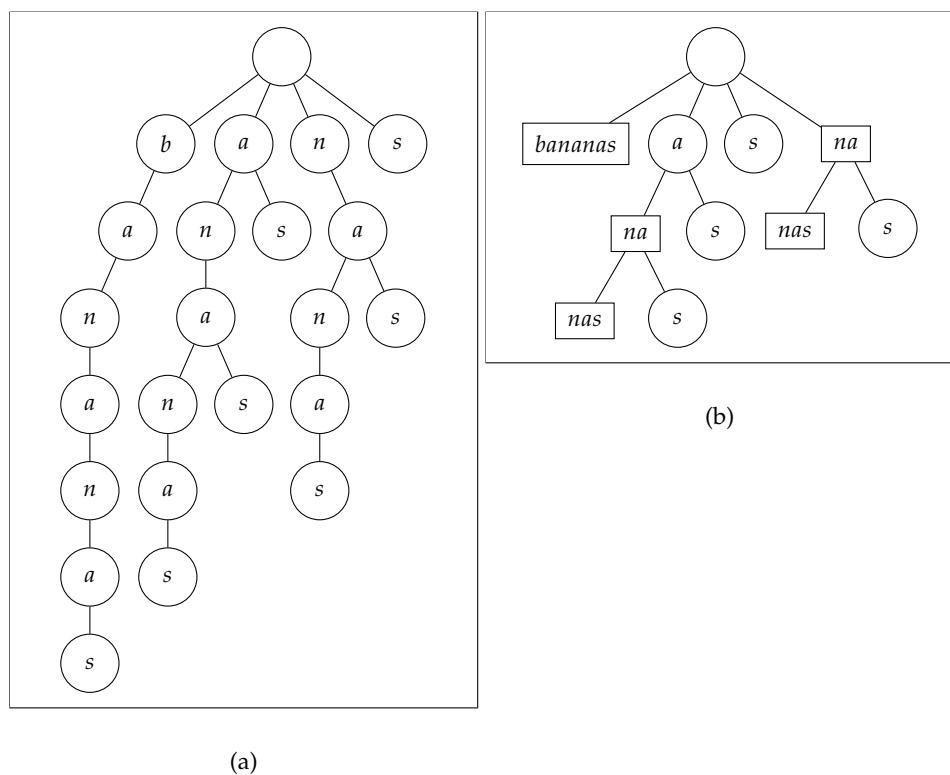


Figure 2.2: The suffix tree (a) and suffix trie (b) of the string “bananas”

2.3 Graph based genome representations

In the previous sections we have been introduced to the variable nature of genetic data. The 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. If we are able to rephrase biological questions in graph theoretical settings, we can also benefit from the extensive mathematical field of graph theory when searching for solutions to the arising problems. However, when changing the underlying structure a major problem appears: In order to avoid a drop in functionality, the more complex model calls for more sophisticated variants of existing methods. 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 contents of the section is presented in a way which should not require any previous knowledge of graph theory beyond elementary terms, but readers interested in a more complete introduction is referred to the bibliography[41, Chapter 9][1, Chapter 11][33, Chapter 0]. Complexity in regards to the graphs and their operations is discussed using *big-O* notation[41, Chapter 2][1, 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 are more complex than single 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 flexible

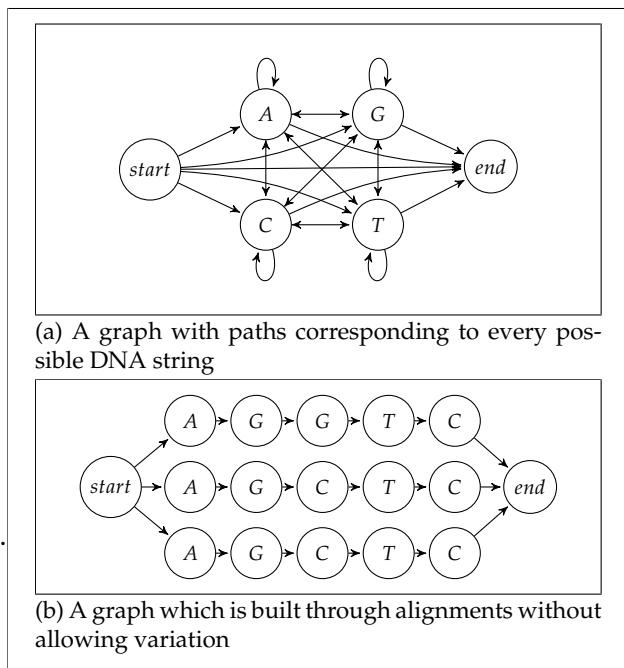


Figure 2.3: Two proposed graph models displaying flexibility (a) and rigidity (b)

they present no consistency, and a flexibility/rigidness-tradeoff becomes apparent (Figure 2.3). How the structures are defined in detailed 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”[29], Pevzner, Tang and Waterman proposes *de Bruijn* graphs as a solution to the problem of finding the correct assembly 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. lets the vertices contain strings of length $l - 1$ and connects vertices with an edge wherever there exists a read of length l containing the two substrings. Formulating the problem in this fashion makes us able to see it as a *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*[1, 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 vertex representing it.

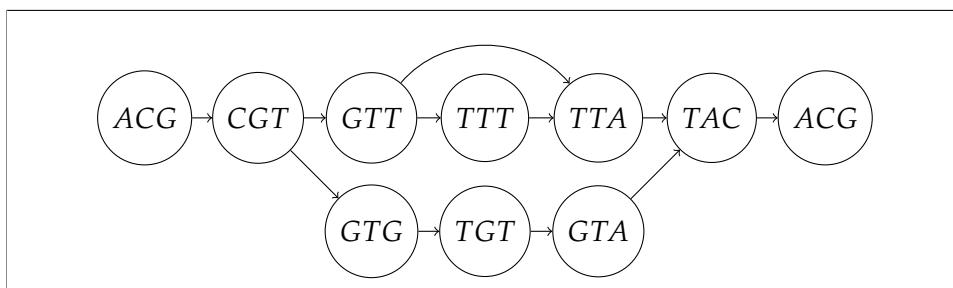
A more detailed type of de Bruijn graphs is the colored variant where the origins of edges and vertices are visualized as colors. The entire sequence originating from a single individual sample can be seen by following a path with a given color. Similarities between samples can be seen as multicolored stretches, variation take 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[12].

Sequence graphs

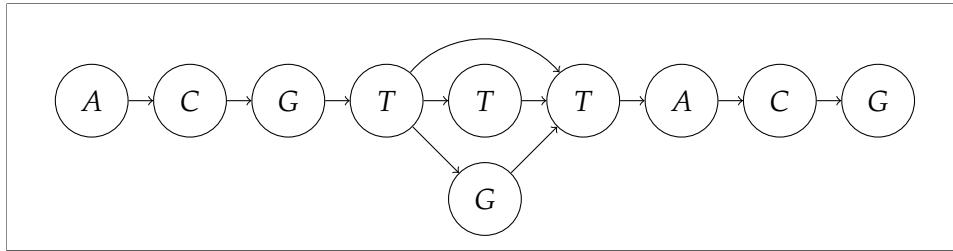
The relationship between a de Bruijn graph and the sequences it represents is not immediately apparent. A more intuitively pleasing representation is a graph where every vertex contains exactly one nucleotide (Figure 2.4b), a concept called *partially ordered graphs* by Lee et al.[6] and *sequence graphs* by Paten et al.[27]. In this representation the underlying connection between the characters of a text string and the vertices of the graph is more obvious. The representation does however have a major drawback compared to de Bruijn graphs: The concept of uniqueness. A vertex 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., for uniqueness. Even though these ids can be used to identify a vertex they contain no information regarding the relationships between the elements. The difficulties presented by this problem will be the basis for the subsequent section on mapping.

Cactus graphs

The article “Cactus Graphs for Genome Comparisons”[28] introduces *cactus graphs* as a model for alignment of multiple genomes. A cactus graph has vertices representing sets of homologous DNA sequences and edges representing adjacencies between the strings in any of the genomes used as input (Figure 2.4c). In cases where there exists several adjacencies between two vertices these are combined into a single edge with several labels. The result is a graph where every *simple cycle*, cycles where no vertex is repeated, has at most one vertex in common. A similarity between this representation and de Bruijn graphs is that the vertices contain subsequences of the original input sequences. Additionally this representations allows some flexibility, controllable through the definition of homology and thus the strictness of “equality” between strings represented in a single vertex. If the strictest possible restriction is set, a requirement of equal strings, the vertices would contain an exact k-mer from an input sequence just like the de Bruijn vertices.



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



(b) Sequence graph

(c) Cactus graph

Figure 2.4: Various graph representations of the strings "ACGTTTACG", "ACGTGTCA" and "ACGTTACG"

2.3.2 Mapping

One of the previously mentioned operations which can control the properties of the model is mapping. Although the two terms are often used isomorphically we will in this thesis define mapping and alignment as two separate concepts. We let mapping be the process of finding relationships between single characters of a string and single elements of a reference genome. Alignment is concerned with finding relationships between consecutive elements of an input string and substructures in the reference genome. In comparison 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. If we assume the indexation system proposed in the last section, the indexes of a graph has only one property: Uniqueness. They do not hold the intrinsic value of describing relations between vertices. Any mapping system which uses fixed coordinates would face problems when dealing with a fluent graph able to 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 nucleotides are the most basic information, there exists an equal number of identically scoring positions for every base as there are vertices containing that vertice in the graph.

Paten et al.[27] 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 going in to the vertice, B is the base contained in the vertice and R is an outgoing path from the vertice (Fig. [figure](#)). More conceptually, contexts can be seen as paths which pass through a given vertice. Because these paths are linear and passes 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 key words left-right and central refer to how a vertice defines its contexts based on the surroundings. The former defines separate contexts for incoming and outgoing paths whereas the latter defines the vertice as a center of a path where the differences of the lengths of the two contexts are minimized. A *balanced central exact match mapping scheme* is a special case of the latter 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.

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 vertice, meaning every

possible context either maps unambiguously to a single vertex or does not map at all. Because 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, which means context-based mapping leads to 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 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 exists 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 results 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”[14] 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 an array-index in the edit distance problems, 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. 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”[26] the previously mentioned concept of context-based mapping is used for aligning entire strings. The algorithm works by identifying sub-strings of the input string which maps 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 graph search through the region represented by the gap between the end and start of consecutive uniquely mapped sequences. 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 of the involved input data.

Chapter 3

Aims

The project in itself had a clear goal: Develop an algorithm for aligning against graph based reference genomes. This thesis will not be concerned with the chronological events of the development process. Instead the thesis will be concerned with presenting the result of the project: The algorithm “Fuzzy context-based search”. 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

In order to succeed with the two smaller goals, a tool was created which implements the algorithm. This tool is available online, instructions on retrieving and using the tool 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 base for later expansions into more specific applications. Some of these might seem as “shortcuts” to the reader: We assure this is not the case. Every time the result of one of these simplifications is presented we defend it. In the later parts of the thesis we reintroduce many of the 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” was published by Novak et al. [26], discussing an approach to alignment which is similar to the one presented in this thesis. The similarities and differences between the two is granted a large part of the discussion section of the thesis.

Chapter 4

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 are 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 6.

In order 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 done in this thesis.

4.1 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 graphs¹ 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 vertex at index i is denoted 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.

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 a way of capturing the combination of several individual characters into text strings 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.

Definition 5 (Full path)

A path P through a graph G where $p_0 = 0$ and $p_{-1} = -1$

¹Discussed in section 2.3.1

There is no correspondence the other way, meaning there can exist full paths which does not originate from a single input sequence. An example of this can be seen in figure 4.1 where a reference graph made from three sequences has 9 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 4.1 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

In an incomplete path there exists 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 have distance 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:

$$\text{pathScore}(P^*) = \sum_{i=0}^{|P^*|-2} \text{gapPenalty}(\text{distance}(p_{*i}, p_{*i+1})) \quad (4.1)$$

where $\text{distance}(x, y)$ 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.

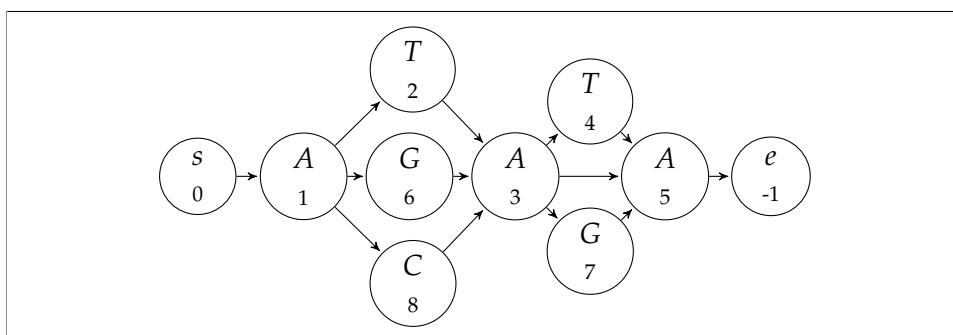


Figure 4.1: An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA"

4.2 The alignment problem

Defining the graphs mean we have a formal notion of half of the input data for the alignment problem as discussed in section 2.3.2. We move on to defining 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 $0 \leq x < |s|$ 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.

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 eachother. 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-vertex, and thus always an invalid mapping. The remaining elements of s is 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 vertex 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 4.1. 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 $A_{x:y} \in A$ spanning the indexes x to $y - 1$ where every element is unmapped. An important aspect here is that every unmapped element should only be considered part of exactly one gap. We cover this aspect by only considering *maximal unmapped subsequences*

Definition 10 (Maximal unmapped subsequence)

A subsequence $A_{x:y} \in A$, such that every $a^* = 0$ 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

$$\text{stringGap}(A) = \sum_{A^* \in A_{MUS}} \text{gapPenalty}(|A^*|) \quad (4.2)$$

where A_{MUS} 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 $0 \leq x < |s|$ with the path score for the incomplete path provided by consecutive mapped indexes of A and the gap penalties for the string gaps in A . We reference this score by $\text{alignmentScore}(A)$.

We can then easily define our graph based adaptation² of the alignment problem:

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 in order to simplify the conceptual explanations of the algorithm. Implementation-wise this can trivially be changed to finding all optimal alignments. The necessary adjustments is discussed as a part of the succeeding chapter in section 5.1.3.

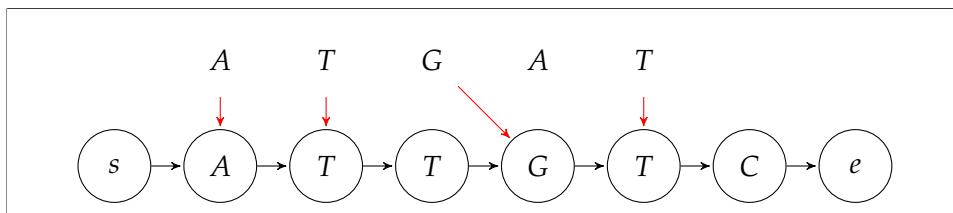


Figure 4.2: A visualization of an example alignment of the string "ATGAA" against a reference genome made from the string "ATTGTC". The actual alignment is visualized through red arrows pointing from the characters of the string to the vertex they are mapped to. We see a gap in the incomplete path between the second and third vertex and that the fourth element of the string is unmapped

²The regular alignment problem for strings was presented in section 2.2.4

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 s *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 numeric value, 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 should be 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 into 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 the 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 do underway.

4.3 “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 set as input parameters, the algorithm assumes a predefind 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 (This should hopefully become apparent throughout this chapter, but a more compact proof can be found in Appendix A).

4.3.1 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 should be a new graph G' with a vertice set V' and an edge set E' .

We first let V' be a subset of the original vertex set V . In order 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 $v_x \in V$ should be in V'_y if there exists an optimal alignment A with a higher score than T where $a_y = x$

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 $v_x \in V$ which is not referenced by a_i in any optimal alignment should not be in V'_i

If we manage to create V' from these two restrictions we can guarantee a vertex set where every element of every optimal alignment is still present and all excesses vertices has been 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. In order 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 vertex $v_x \in V$ should be in V'_i for every $0 \leq i < |s|$.

This is a complete relaxation and puts every vertex $v \in V$ in every subset of V' . The resulting parenting candidate vertex 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

We let a vertex 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 al³. 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 represents the paths passing through it, and because we know that if a context is

³Context-based mapping is explained in detail in section 2.3.2

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 $0 \leq i < |s|$ 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 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 direction. Because we allow gaps in our alignments we have to extend the number of steps a vertex looks back 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 $0 \leq j < i$. 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 the latter is an integer

We let the weight w denote the distance $d(P)$ of the shortest path P where $P_0 = i_s$ and $P_{-1} = i_e$. These weights can be found through a regular graph search in G , if no distance is found we let the value be inf. At this point we have a complete graph G' .

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 $e' = \{i_s, i_e, 0\} \in E'$

The resulting graph is still very complex. Every vertex is connected to every preceding vertex, and in order 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 incomplete paths which have a lower score than T and we thus can limit the edges to only the ones that are passable without being penalized more than λ . This creates an upper bound both on how far back in the candidate sets a vertex looks for neighbours and, more importantly, the complexity of the individual graph searches done in G .

4.3.2 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 align-

ment 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-MSA⁴, 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 candidate vertices from the reference 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 node in the last candidate set.

The base case of the dynamic programming are the candidate vertices in the first candidate set, $v_x \in V'_0$. We initialize these scores to $\text{mappingScore}(b(v_x), s_0)$, 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, for instance $-2 * \lambda$, which yields any alignment starting with the vertex a score lower than T . This means they can not be considered candidates for the optimal alignment.

The recurrence relation of the dynamic programming algorithm is concerned with setting the score for any vertex/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 edges, find the one yielding the highest score and add $\text{mappingScore}(b(v_x), s_y)$. The score for an edge is found by taking the score in the vertice $v_y \in V'_j$, represented by the start-index i_s in the edge, 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 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

⁴The DP algorithm developed in [14] presented in 2.3.3

scoring high enough, meaning the path is never interesting when creating 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 can simply deem the string as unalignable.

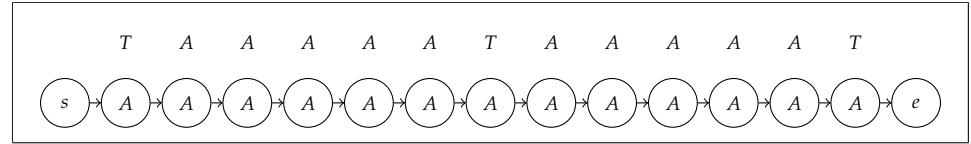
4.4 Heuristical applications

In this chapter we have presented a strict non-heuristical algorithm which is guaranteed to give the correct results whenever it is fed the correct parameters. This is a concept which provides formally steady ground when discussing the tractability of the conceptual solution to the problem. When nothing else is specifically stated, the remainder of the thesis will be concerned with the approach presented up to this point, in all its formally provable glory. However, we now want to take a brief detour to examine some of the heuristical possibilities which emerges. In the approach, if we are not able to find a good enough score for a vertex in the last candidate set we simply classify the string as unalignable. This might seem a bit rash, as we have already completed a strenuous search for similarities between the two structures. As previously mentioned this is a design choice taken to land at an easily perceivable, formally strict, proof-of-concept algorithm. If one were to imagine the strict optimality restriction were dropped, it is fair to assume that there exists a set of "good" solutions in the combination of the entities we have found in the search. We can see one such example in figure 4.3a: Because there are three mismatches the regular approach would need an error-margin $\lambda \geq 3$ to find the alignment⁵. If we set the context length $|c| \leq 2$ there are no vertices which has more then a single mismatch in their contexts. Because of this we are able find all the candidate vertices, and the path through them, for the optimal alignment with $\lambda = 1$. The algorithm will however drop it as a solution because of the alignment score dropping below the threshold. We propose a heuristical approach would present the found, and in this case optimal, solution. The technical details on how this can be implemented is described in section 5.2 of the next chapter.

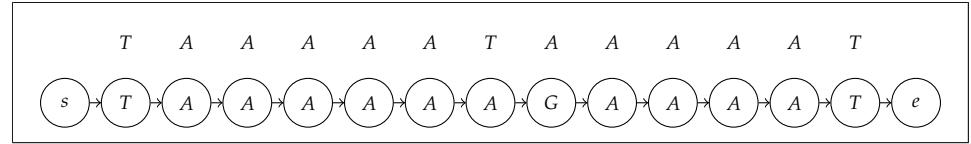
There is a specific reason for devising an algorithm which sometimes discards the optimal solution. In figure 4.3b we can see an alignment with two mismatches. If we let set $|c| = 2$ and $\lambda = 1$ there will be candidate vertices which are dropped due to pruning in the initial context search. If we imagine the two reference graphs in the figure are actually two different

⁵Assuming regular edit distance scoring

paths in the same graph, this will result in the heuristical algorithm we propose discovering the path with three mismatches and not the one with two, thus ending in a non-optimal alignment. If we utilize different error margins for the first and the second step of the algorithm, it is no longer given that the optimal path through the candidate graph is equivalent to the best path through the original graph. For the approach proposed up to this section we have shown guaranteed optimality by setting some constraints. This is the reason, when nothing else is specified, we will be concerned with the non-heuristical algorithm for the remainder of the thesis.



(a) A sequence with three apparent SNPs aligned against a reference graph



(b) The same sequence aligned against a different reference graph, showing two apparent SNPs

Chapter 5

Implementation

In this section we will present the implementation of the algorithm “Fuzzy context-based search” which is found in the *GraphGenome* tool (Appendix C). The algorithm will be coupled by a explicit example found in figures 5.1-5.3 and table 5.1. 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 problem¹ 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, we will briefly present some possible variations on the implementation which is referenced in the remaining chapters of the thesis.

5.1 Aligning sequences

The alignment process consists of the two steps described in the previous chapter, which each has 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.

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

5.1.1 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, is generated for every vertex 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 to the context set $c(v_i)$ for every vertex 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 vertex v_x is popped from the top of the queue the first operation consists of checking whether the context set of the vertex is done. A context set is complete if every vertex in the incoming neighbour set $n_i(v_x)$ of the vertex is marked as finished. If the context set is not complete the vertex reinserts itself at the end of the queue. In the opposite case, when a vertex is deemed as ready, it starts generating contexts for its outgoing neighbours $n_o(v_x)$. The vertex 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 vertex 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 vertex is marking itself as finished.

In order to avoid making the end vertex 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 vertex along every path leading up to t_G has generated its context, and as we know every vertex stems from an input sequence and every input sequence ends in t_G we know every vertex 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 and the algorithm starts again to generate right contexts.

As sets per definition do not allow duplicates the impact of a branching occurring in the graph will fade away after exactly $|c|$ steps as the difference is trimmed away (Figure 5.1), and thus avoid explosive exponentiality in the context set sizes. Unlike the method of Paten et al. [27] there are no requirements for contexts to be uniquely mappable to exactly one vertice. Because the last step of the algorithm does a 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 it's originating node as a value (fig. 5.2). 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 $b = 1$ and thus $b^{|c|} = 1^2$. The current implementation uses a naive suffix tree implementation where insertion is $O(|c|)$, giving a total time complexity of $O(b^{|c|}|c|)$ per node per context set and $O(2|G|b^{|c|}|c|)$ for the entire graph. The generation process visits $|G|$ vertices once in each direction, which means the entire index can be built in $O(|G||c|)$ ³.

²Actual computations for b can be found in section 7.5

³Assuming $b^{|c|} = 1$. Further explanations of simplifications done in big-O notation can be found in the bibliography[41, Chapter 2][1, Section 3.1]

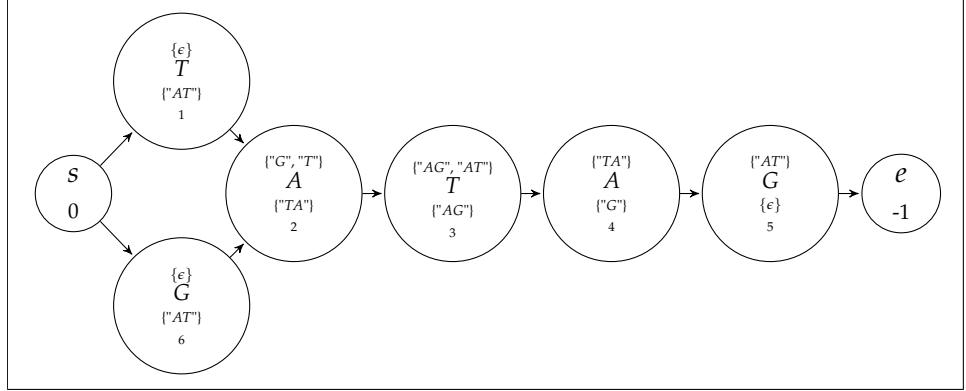


Figure 5.1: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

5.1.2 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:

$$scores_b[i] = \max \begin{cases} scores_b[i - 1] + gapPenalty(1) & \# \text{ String gap} \\ scores[i] + gapPenalty(1) & \# \text{ Graph gap} \\ scores[i - 1] + mappingScore(c_i, b) & \# \text{ Mapping} \end{cases} \quad (5.1)$$

An important aspect is that this procedure uses the same scoring schema as the one defined for the entire alignment. This newly created array $scores_b$ is then supplemented to the same recursive function in the child as the main array $scores$. When a leaf vertex is reached the last index of the supplied

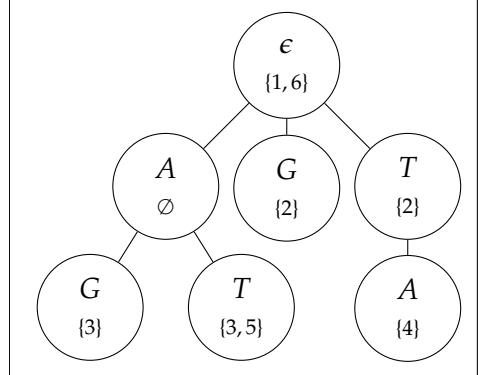


Figure 5.2: The left suffix tree corresponding to the graph in 5.1

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 lot of matches. The tool 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 $|l| < 2 * c + 1$ 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 atleast 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.

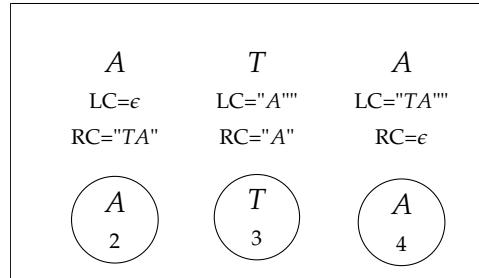
After the fuzzy search is concluded there are two maps of candidates for every index, one containing the vertices matching the left context and an equivalent for vertices matching the right context. The keysets of these maps are intersected to produce a final candidate set for the index i , where the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

⁴See calculations in Appendix B

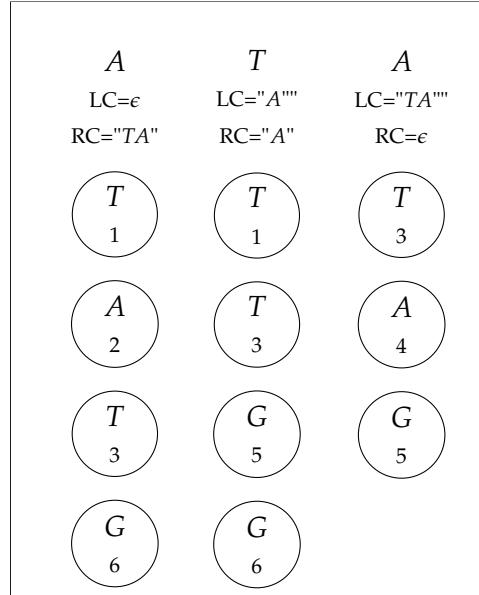
as an ordered list of sets V'_i for $0 \leq i < |s|$ where

$$V'_i = \{v_x | v_x \in G \wedge \exists [c \in c(v_x)] (\text{contextScore}(c, c(s_i)) \geq T_{c(s_i)})\} \quad (5.2)$$

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



(a) $\lambda = 0$



(b) $\lambda = 1$

Figure 5.3: The resulting candidate sets for mapping the string "ATA" against the reference genome from fig. 5.1 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.

5.1.3 Searching the new graph with a modified PO-MSA search

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.

Table 5.1: The 4 arrays used by the searching algorithm when using the candidate sets from Fig 5.3 and $T = -1$

index in the three separate arrays reference the same vertex.

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

$$\begin{aligned} \textit{indexes}[0][j] &= x \\ \textit{scores}[0][j] &= \textit{mappingScore}(b(v_x), s_0) \\ \textit{backPointers}[0][j] &= -1 : -1 \end{aligned}$$

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

searching algorithm when using the candidate sets from Fig 5.3 and $T = -1$ 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 $\text{indexes}[i][j]$ 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 $\text{indexes}[i']$. 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, $\text{scores}[i'][j']$, the gap penalties, and a mapping score for the current index $\text{mappingScore}(v_{\text{indexes}[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length $\text{distance}(v_{\text{indexes}[i'][j']}, v_{\text{indexes}[i][j]})$. 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 vertex $n_{\text{indexes}[i'][j']}$ and is concluded when we find $n_{\text{indexes}[i][j]}$. Whenever we search for more than λ steps without finding the target vertex, 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 vertex pairs ending in the vertex with index $indexes[i][j]$. $backpointers[i][j]$ is set to the index-pair (i', j') responsible for producing this score. The recurrence formulas for

the three arrays are thus:

$$\begin{aligned}
 \text{indexes}[i][j] &= x & n_x \in V'_i \\
 \text{scores}[i][j] &= \max_{i', j'} \theta((i, j), (i', j')) & 0 \leq i' < i, 0 \leq j' < |\text{scores}[i']| \\
 \text{backPointers}[i][j] &= \operatorname{argmax}_{i', j'} \theta((i, j), (i', j')) & 0 \leq i' < i, 0 \leq j' < |\text{scores}[i']|
 \end{aligned} \tag{5.3}$$

where θ is a scoring function defined as:

$$\begin{aligned}
 \theta((x_1, y_1), (x_2, y_2)) &= \text{scores}[x_2][y_2] \\
 &+ \text{gapPenalty}(x_1 - x_2) \\
 &+ \text{gapPenalty}(\text{distance}(n_{\text{indexes}[x_2][y_2]}, n_{\text{indexes}[x_1][y_1]})) \\
 &+ \text{mappingScore}(b(n_{\text{indexes}[x_1][y_1]}), s_{x_1})
 \end{aligned} \tag{5.4}$$

When the iteration ends we will have a score for every candidate vertex in every candidate set. We can iterate over the scores corresponding to the last candidate set, $\text{scores}[|s| - 1]$ to find the highest alignment score. The optimal alignment ends in the vertex 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 $O(\frac{\lambda^2 |s| (|c|^\lambda |G|)^2}{4^{|c|^2}})$ (Appendix B). 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 on only 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.

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.

5.1.4 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 (λ is not penalized more than λ), but the combination is not good enough, or the path goes through a step in which there are no legal possibilities for traversal. The last would 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 paths scoring high enough and both result in an *empty alignment*, an alignment where every element of s is unmapped.

Yielding an empty alignment for the first case might seem strange as the algorithm does a full computation and finds a legal path. There is however one important consideration: Vertices which are parts of an optimal path might have been dropped during the pruning of the candidate sets. There is one very interesting case where this occurs, namely when the difference between the threshold T and the highest scoring alignment is exactly 1 (or the minimal penalty possible for generalized context schemes). In this case we know there are no paths which yield a score of T , because we would have found them, and there are no unfound alignments scoring higher than the one we found because there exists no such possible scores. In order to stay in line with the strict problem definition these cases are also classified unmappable.

The second case is often less interesting as it usually leads to vastly less coherent alignments and as a result typically lower scores. Even so, there are interesting concepts touched upon also here: Because of the distinct penalty given to invalid edges we have no real way of knowing how the proposed alignment is really scored. Thus it can, much like the case discussed above, contain interesting regions which are aligned well within the threshold.

5.2 Necessary adjustments for a heuristical search

These interesting regions contain the information we would want to use when creating heuristical alignments. The process is mainly dependant on not doing the hard classification as unalignable, but instead return the best option we can find. There are some additional changes necessary: First off we want to make sure we tune the constant penalty for gaps so large we don't consider them in a way where they do not take priority over actual paths. This can be done by increasing the arbitrary value

to another large number, for instance $gapPenalty(|s|)$. If we want the heuristic algorithm to find incomplete paths which does not previously exist in the graph, by jumping between branches or horizontal positions, we would need to devise a penalty for non-existent gaps. This could be a process potent when aligning sequences which partially cover large structural variations. A final tuning necessary for the heuristical approach is to allow the last mapped element of A to occur before the last index. This is done by iterating backwards over the candidate sets at the cost of a gap penalty. These modifications are available in the implementation through the `--heuristical=true` parameter.

5.3 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 extremely trivial parallelization of this step is implemented through splitting up the indexes into separate threads, reachable through the `--parallelization=true` flag. Further parallelization 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.

5.4 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 decisions 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 5.4). Too avoid confusion we have split the two processes completely apart, letting the alignment algorithm do all the heavy lifting to keep the merge as as much of a straight forward procedure as possible.

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 representations 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.

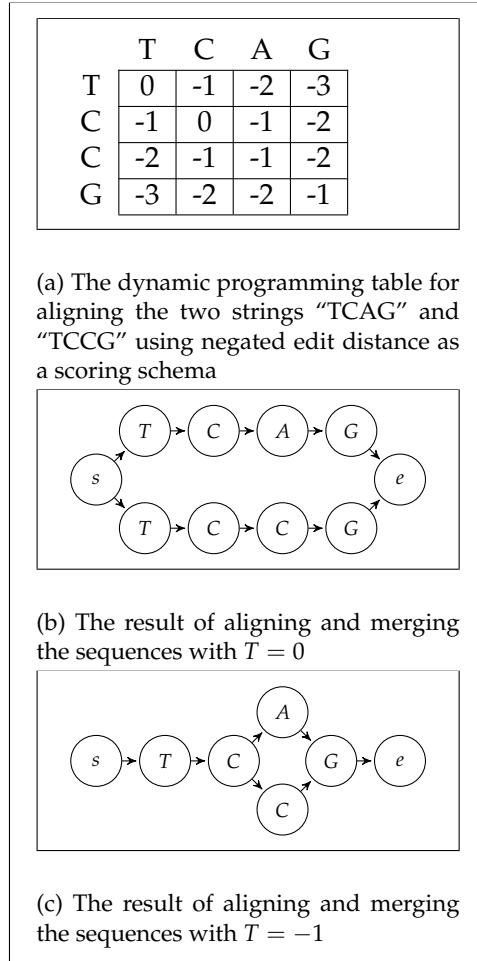


Figure 5.4: Different scoring thresholds T yields different reference graphs

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

$$eq(s_x, v_y) = \begin{cases} True & If a_x = v_y \\ False & Else \end{cases} \quad (5.5)$$

When we have this way of checking equality between the two graphs we can simply let the vertex set V^* of the merged graph be the union of the original vertex set V and the vertex set of G_s , where every “new” vertex 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 vertex 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 vertex $curr$ if $a_i = 0$ and thus unmapped. We also create a new vertex if index is mapped, but the characters s_i and $b(v_{a_i})$ is not equal. This vertex 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 vertex v_{a_i} . At this point we have a pair of vertices, $prev$ and $curr$, which represents 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 vertex 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: Alignment of short reads or a combined operation of aligning and merging complete sequences. This is important to point out, because we in the problem definition state that the algorithm is tuned for aligning reads shorter than the reference. As previously stated 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 assembly process, where smaller reads are aligned, combined and eventually merged. This assembly process does not need to diverge from known the standard assembly which deals with linear references, as the indexes found in alignment are unique and can be used by overlap-techniques or as a basis for de Bruijn graphs.

Chapter 6

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 now increase the level of abstraction from the realm of formal details into "What results can we expect from using this 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 it is we actually deem as correct behaviour, and how this behaviour is manifested through the input/output pairs. Throughout the development and implementation process these expectations and many more have been used as requirement specifications, realized through a test set which can be found in the github repo¹.

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.

6.1 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-

¹Found in the presentation of the tool in Appendix C

tions can be nested. This is a step which is crucial 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.

6.2 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:

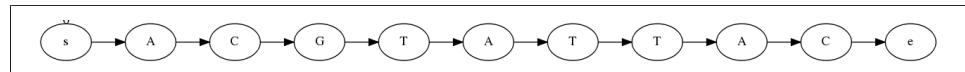


Figure 6.1: The reference graph made from the sequence "ACGTATTAC"

At last 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 particularly 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 to 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
```

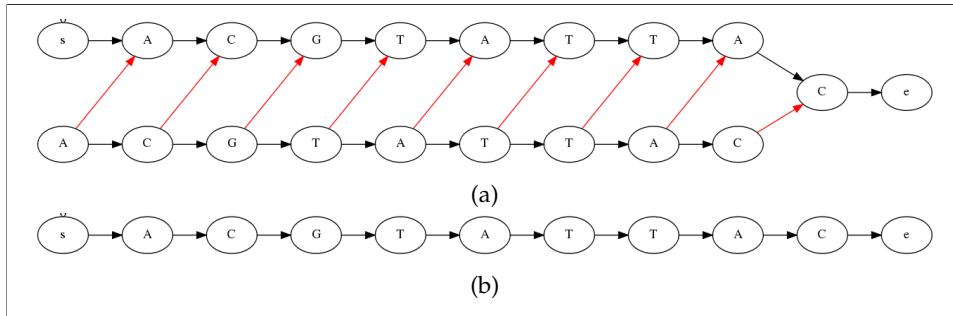


Figure 6.2: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 6.1

- 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 be 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
```

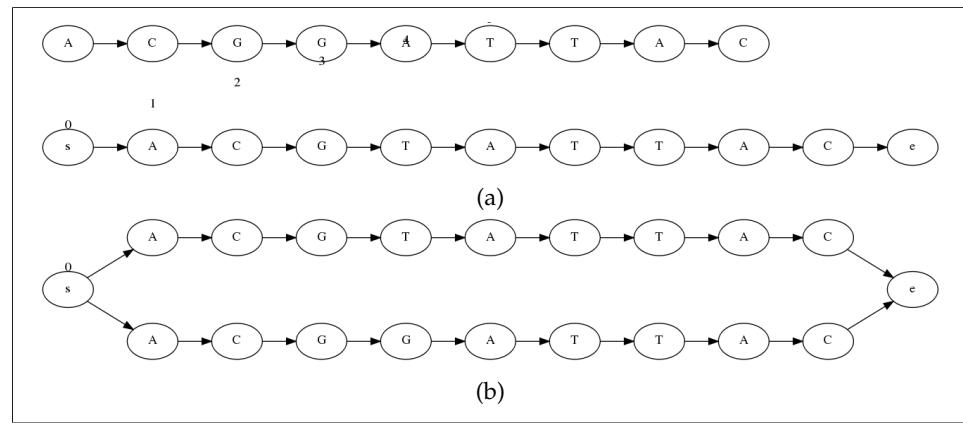


Figure 6.3: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 6.1

- 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
```

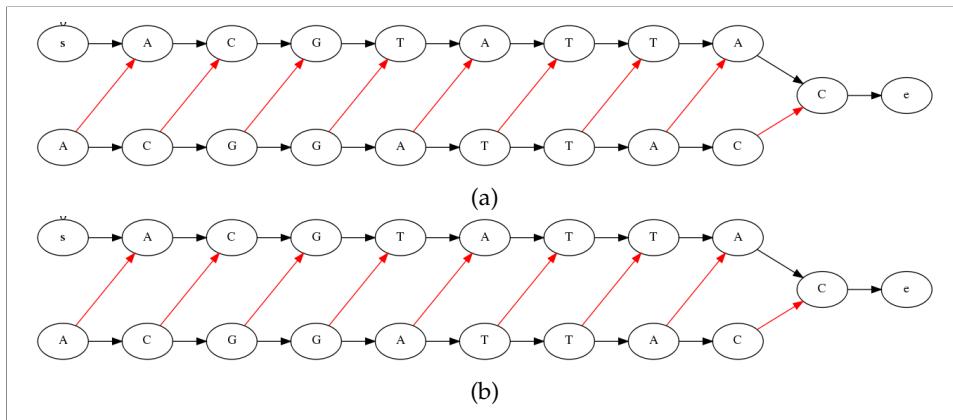


Figure 6.4: The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference seen in figure 6.1

- 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 vertex more than the old graph, where exactly one vertex has one more outgoing edge and exactly one vertex 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
```

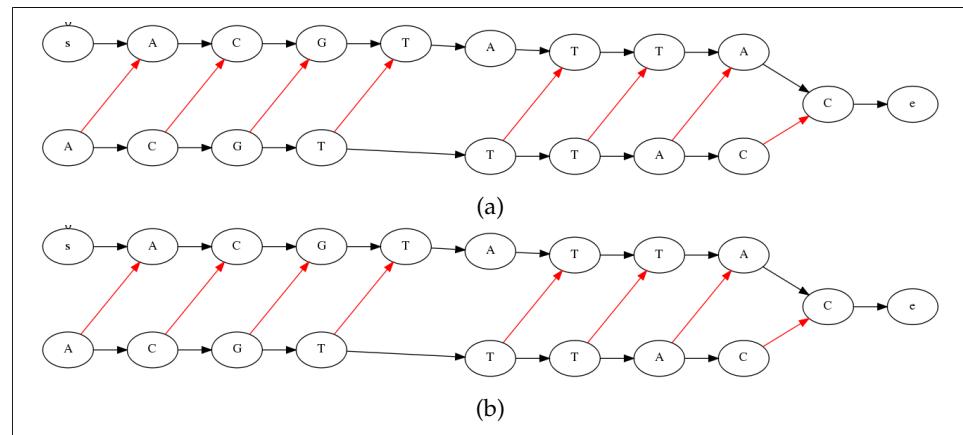


Figure 6.5: The result of aligning (a) and merging (b) the sequence "ACGTTTAC" against the reference seen in figure 6.1

- 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
```

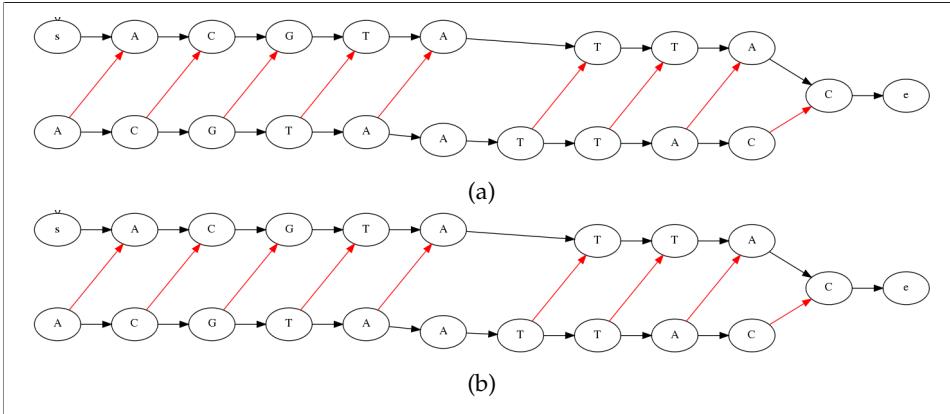


Figure 6.6: The result of aligning (a) and merging (b) the sequence "ACGTAATTAC" against the reference seen in figure 6.1

- 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 vertex 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 gaps². In the case of a large insertion we will see reads which obviously does 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 assembly process³. 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 an assembly 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

²These are discussed as a part of the heuristical approach in 4.4 and 5.2

³Briefly mentioned in 5.4

is thus not shown.

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 flexibility 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
```

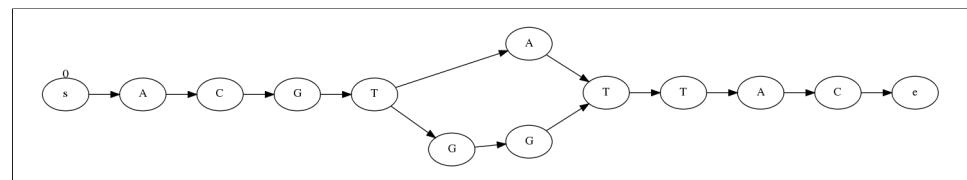


Figure 6.7: The result of merging the sequence "ACGTGGTTAC" with the reference seen in figure 6.1

- 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 fairly more vague then previously. This is not because the approach lacks precision. It is moreso because we are not sure whether our intuition is precise, and if so whether the precision handles ambiguous cases equally. 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
```

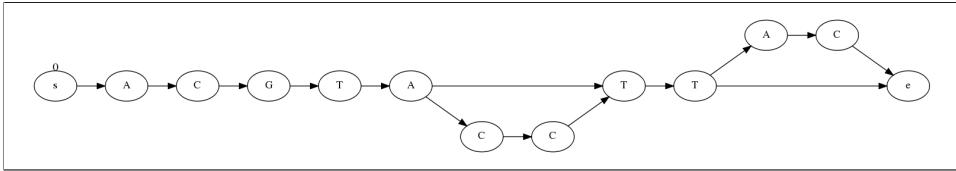


Figure 6.8: The result of merging the sequence “ACGTACCTT” with the reference seen in figure 6.1

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. It is suboptimal that this kind of ambiguity exists, but it is a burden carried by the scoring schema and the problem definition rather than the approach itself.

Complex graphs

In 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
```

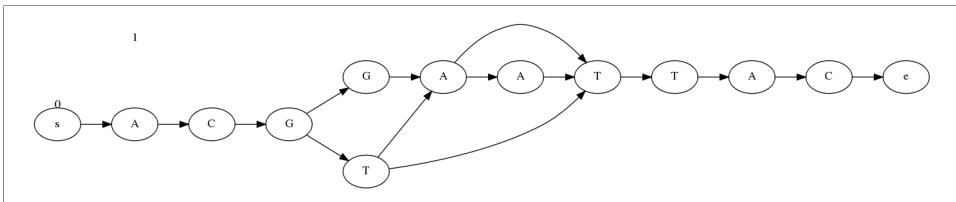


Figure 6.9: The reference graph made from the sequences “ACGTATTAC”, “ACGGATTAC”, “ACGTTTAC”, “ACGTAATTAC”

- 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
```

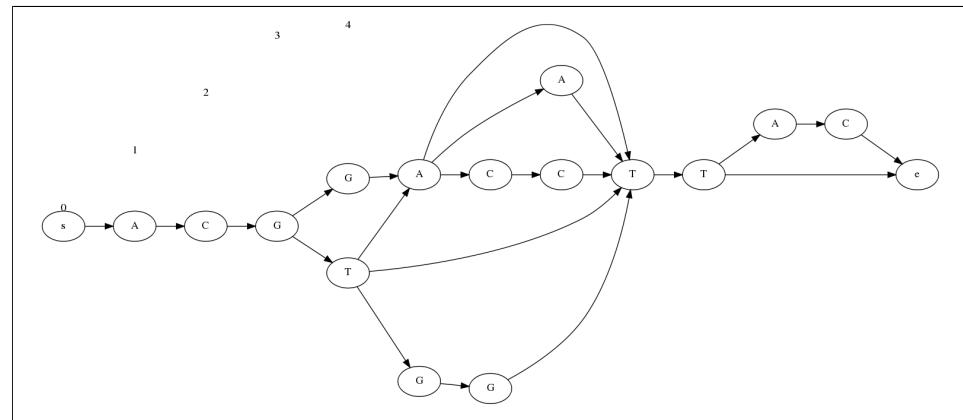


Figure 6.10: The two sequences “ACGTGGTTAC” and “ACGTACCTT” merged into the graph reference from figure 6.9

Although it is small, this is what we would consider to be an extremely complex graph in the domain of genetic information. **The value of the graph based approach becomes apparent through the visualization: The variable genomic regions present themselves in a way which begs to be analyzed.** 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 can 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

```

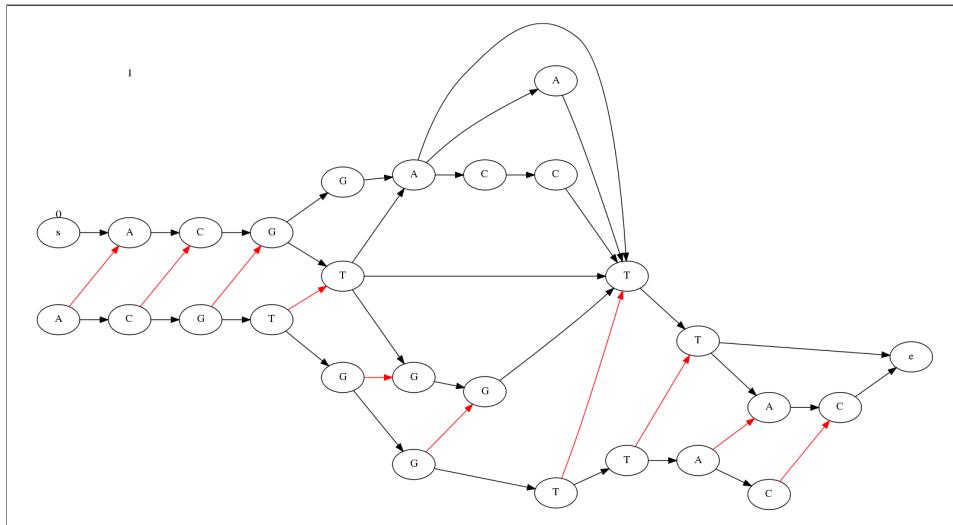


Figure 6.11: Aligning the sequence “ACGTGGTTAC” against the reference graph seen in figure 6.10

Because of the quick growth in complexity we will 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 7

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. How correctness is determined is covered in section 7.2. Most of the results are compared against running the same alignment with an own PO-MSA implementation. 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.

7.1 Test data

These tests are meant to reflect usage in what would be an every day situation, and does therefore use real genetic data. All of the sequences are FASTA-files from the MHC region fetched from the vg github repo[39] and the test-set provided by the sequence graphs tool[25]. The exact test sets are chosen to provide a variety of sequence lengths. In order to cover lengths where we found no sequences there are created artificial sequences by cutting out suitable regions from longer sequences. All the sequences which are used can be found in the test-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[23] 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 seen in the appendix. A read r from a graph G is generated by the following procedure:

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.

Because this thesis is concerned with the mathematical properties of the model the noise in the reads does not necessarily depict the true nature of either genetic variation (Section 2.1.2) or read errors (Section 2.2.3). This is one of the points where we have chosen to limit the complexity of the problem. **As long as we see the approach as a solution to the most general problem we argue this is an acceptable limitation.**

7.2 Validation

When an alignment is produced for a read we classify it as either as correct or not correct. Intuitively one could imagine this can be figured out by determining 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 mathematical 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. This is because of the non-heuristical properties of the algorithm: We are able to devise parameters and test data in such a way that correctness is not an interesting measurement. In all tests where accuracy is omitted this is because the approach had a 100% success rate.

7.3 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 unnecessary 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 7.6. The time unit used throughout the chapter is milliseconds.

7.4 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 7.1. All of these are linear operations, a trend which is clearly visible. The tool was not able to build an index for the largest input file with 4.5mb because of insufficient memory.

When splitting up the runtime into the individual constituents an interesting pattern emerges: The main load of the indexation lies in writing to file (Figure 7.2). The data structure used is a large tree of nested complex structures, which the standard Java serialization does not handle well **PROB SHUD REFERENCE SOMETHING. API MAYBE**. Putting effort into improved serialization or reducing the size of the index will thus have a dramatic effect on the time complexity.

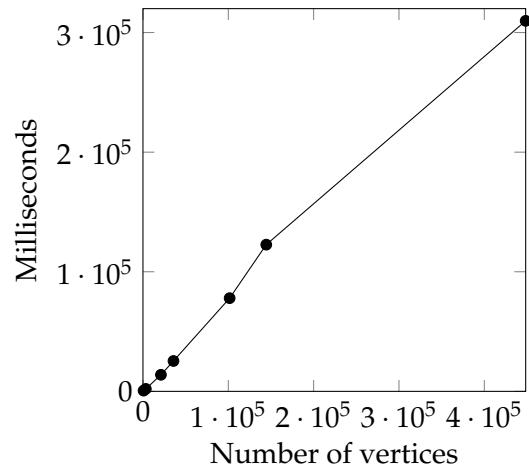


Figure 7.1: Runtime for the build index procedure

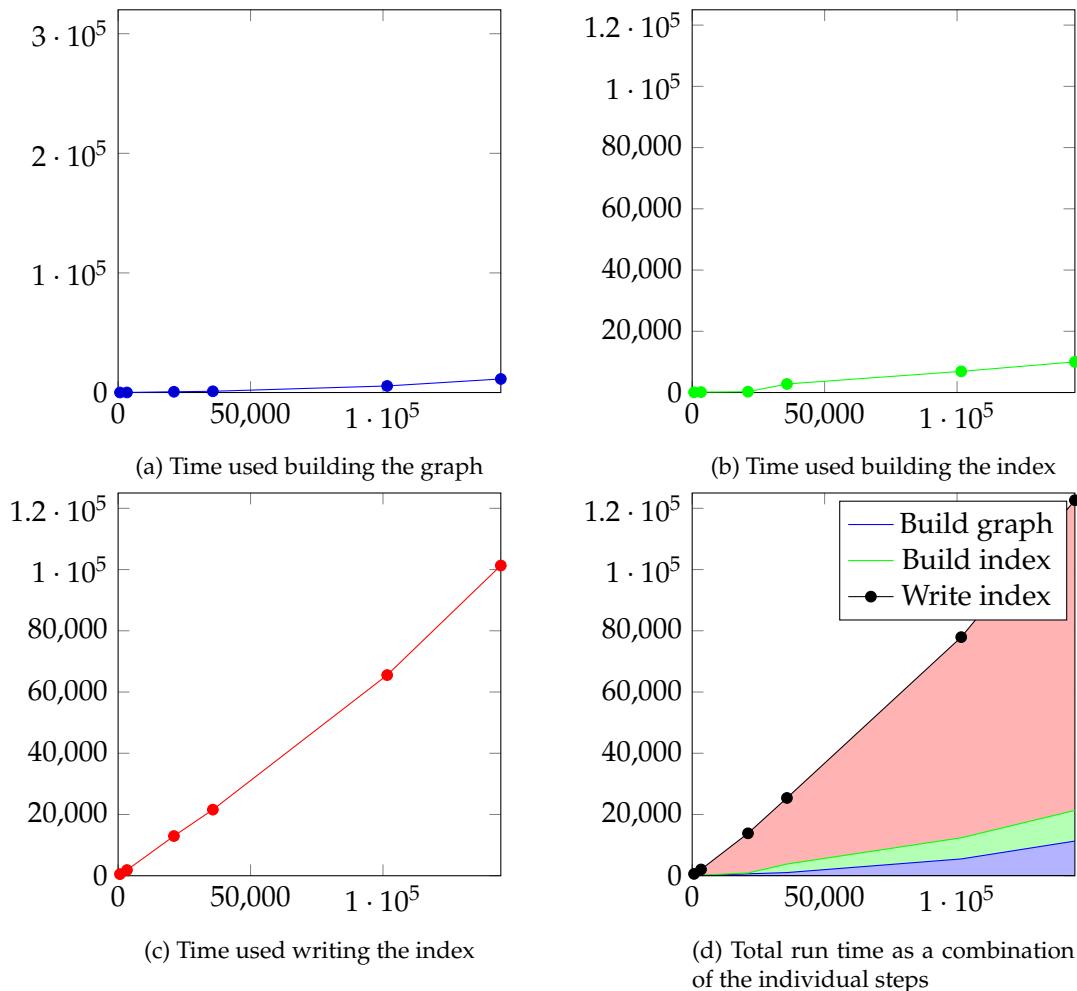


Figure 7.2: Time used by the indexation process as a function of the number of vertices

7.5 Alignment

The alignment tests are run by the align_sequence.sh script, both with --type=fuzzy and --type=po_msa parameters. The section is divided into segments, based on what variable is tuned. 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

Additionally we include one more variable:

- 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 “non-important” variables are locked to a standard value:

- $|G| = 35.000$ Representing the mid-range of our test-sets.
- $\lambda = 0, p = 0.0$ We let alignment back to the origin represent the base case in the study.
- $|s| = 120$ Common read length for the Illumina HiSeq3000/4000 technology.
- $b = 1$ Calculations can be found in section 7.5 which is concerned with tuning this variable.

Runtime as a function of graph size

We start by comparing the two alignment algorithms on different graph sizes. Figure 7.3 shows the averaged results over 50 runs of each algorithm on the smallest 6 datasets. The two largest datasets were omitted to avoid blowing the scale. As expected PO-MSA is completely linear with regards to the number of vertices. The fuzzy search is only dependant on the graph size to decide the depth of the suffix tree which needs to be searched in the first step, and the extremely logarithmical relationship comes across almost as constant in comparison.

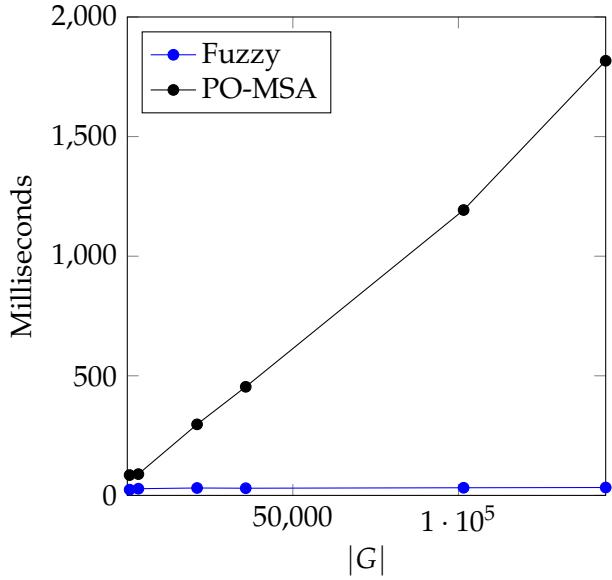


Figure 7.3: Runtime of the alignment process as a function of $|G|$

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 7.4. We can see an exponential growth, which is expected from the complexity analysis¹. Disconcertingly, the algorithm quickly overtakes PO-MSA, already at $\lambda = 2$ the straight forward search is more efficient. Utilizing the results from the previous experiment we can identify that the starting point of the exponential growth is dependant on the graph size. Figure 7.5 shows the same set of tests on the larger graph, where we can see that the approach is more efficient up to $\lambda = 2$. In the same figure we can see that the naive parallelization can soften the exponential growth by a linear factor, at the cost of an initial overhead.

It is important to point out that the extreme growth in relation to the small numbers is in part determined by the flat scoring system. The exponential growth is related to the number of possibilities given λ , not necessarily λ itself. A more fine grained scoring schema might produce a less explosive growth.

¹Found in Appendix B

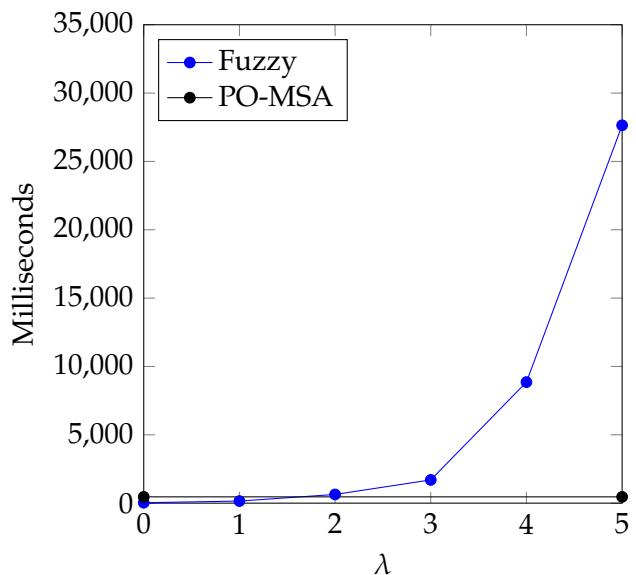


Figure 7.4: Runtime of the alignment process as a function of λ

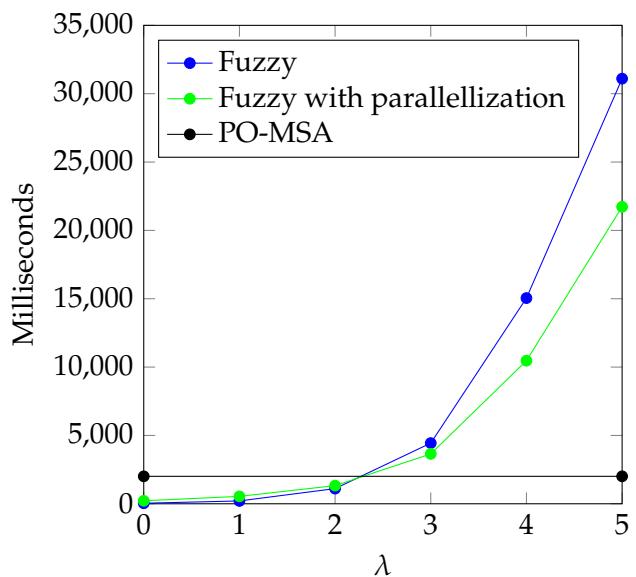


Figure 7.5: Runtime of the alignment process as a function of λ with $|G| = 150.000$ and --parallelization=true

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 taken from a set of sequencing machines, chosen to portray a diversity of read lengths^{ref}. Both the technologies, the lengths and the runtimes are listed in table 7.1. The numbers are visualized in figure 7.6

Technology	Read length	PO-MSA time	Fuzzy time
HiSeq2000 (min)	50	171	19
SOLiDv4	100	294	36
HiSeq3000/4000	120	0	0
Ion PGM	200	676	59
Sanger 3730xl (min)	400	1117	98
454 GS FLX	700	1600	167
Sanger 3730xl (max)	900	2121	194

Table 7.1: Running times for different read lengths for the PO-MSA and fuzzy algorithms

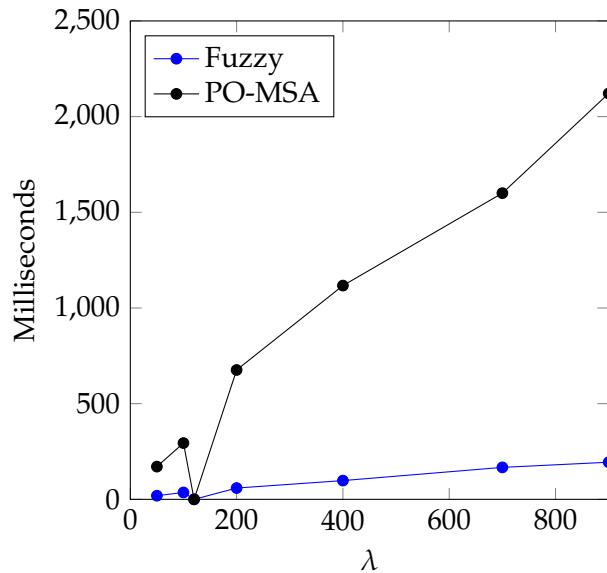


Figure 7.6: Runtime of the alignment process as a function of $|s|$

Runtime as a function of graph complexity

We let the branching probability b denote the complexity of our graph. We generated vcf-files 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 7.7. As seen in the table, the experiment was run on rather moderate values. If we use the numbers from section 2.2.2, assuming all variants are singular and specific to a single individual, a population graph of the human genome would have a branching factor $b = 1 + |p| * 0000076$, where $|p|$ denotes the number of individuals². In this case the largest branching factor presented in the figure would cover a a population of 10.000. In the far more probable case of overlapping and complex variants the number will be larger. **Needs more tests and more motivation behind the sizes.** It is important to remember the number of variants present in the graph is an exponential combination of both b and the graph size.

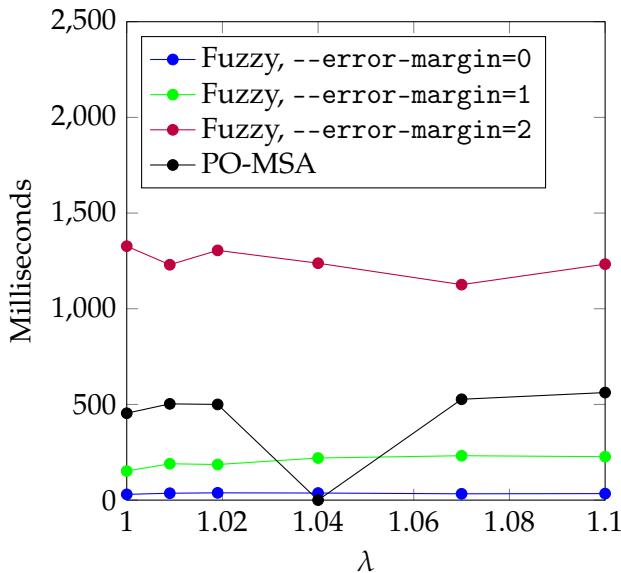


Figure 7.7: Runtime of the alignment process as a function of b

²23.000 variants / 3.000.000.000bp

Correctness 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 100 runs with a variety of settings is seen in figure 7.8. The regular algorithm, seen in 7.8a, shows a clear linear relationship between p and λ . This will be the first set of tests where we also include the results from the heuristical algorithm, shown in 7.8b

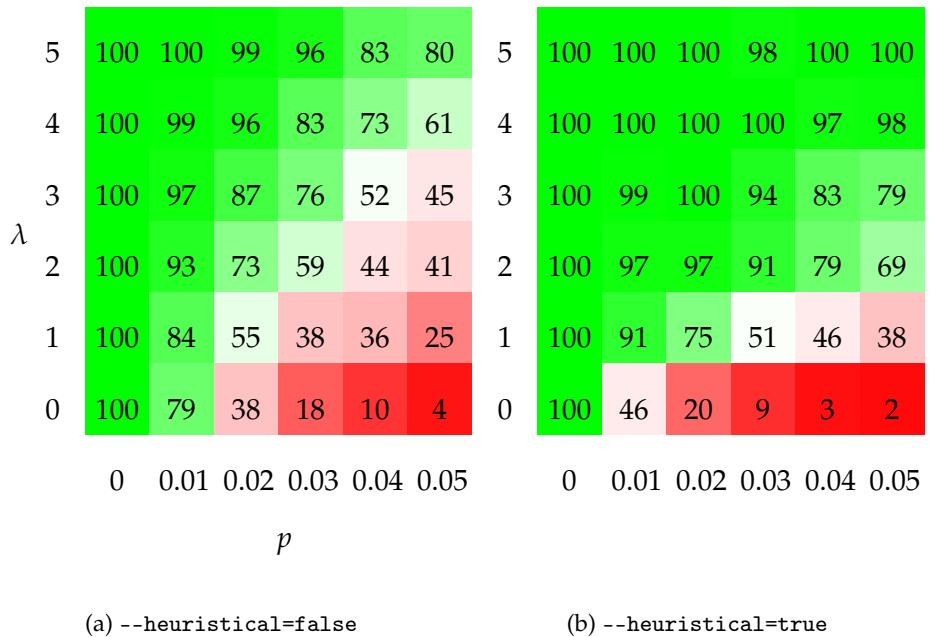


Figure 7.8: Percentage of correctly aligned reads as a function of both p and λ varies

7.6 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". This might seem like a section which should have been granted more space in this chapter. There are several specific reasons this is not the case: First off the tool is listed as unfinished by the creators on their github page. Secondly, we only got the tool running on graphs a fraction of the size compared to the remaining tests in this section. Lastly we previously discovered a large overhead following the serialization of the index, a process we have not focused on in this thesis. The time comparison between the tools are found by timing the entire execution, where we see a lot of room for overhead such as this. However, the comparison is still included as we hope to see underlying factors which can create a setting for the conceptual comparison done in section 8.3. The two tools were compared in building the index and doing an alignment, the results can be seen in respectively figure 7.9 and 7.10. Because the sg tool has limitations with regards to graph size, we have introduced several smaller sample fasta-files found in their test-folder as input data. The time spent by our tool has been divided into "functional parts" and file I/O to indicate the amount of overhead expected in the operations.

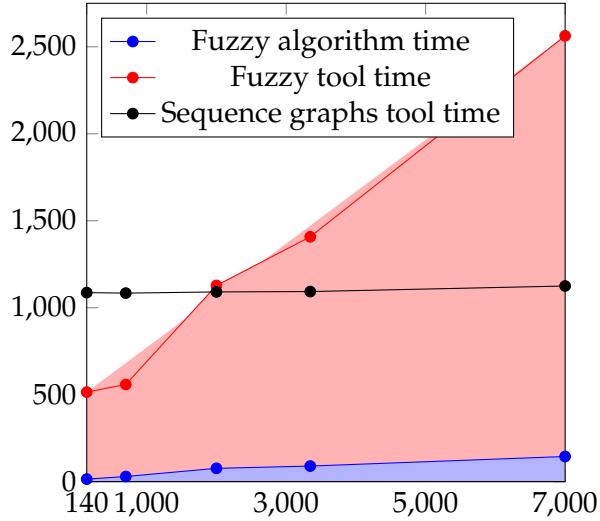


Figure 7.9: Time spent building the index by the different tools

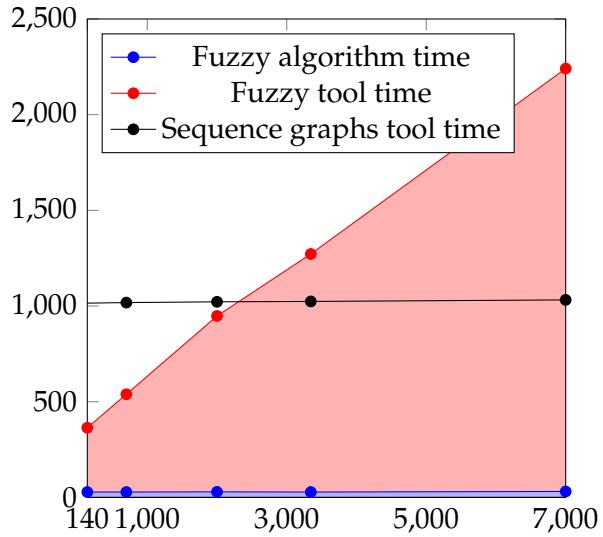


Figure 7.10: Runtimes of the two tools

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 7.11.

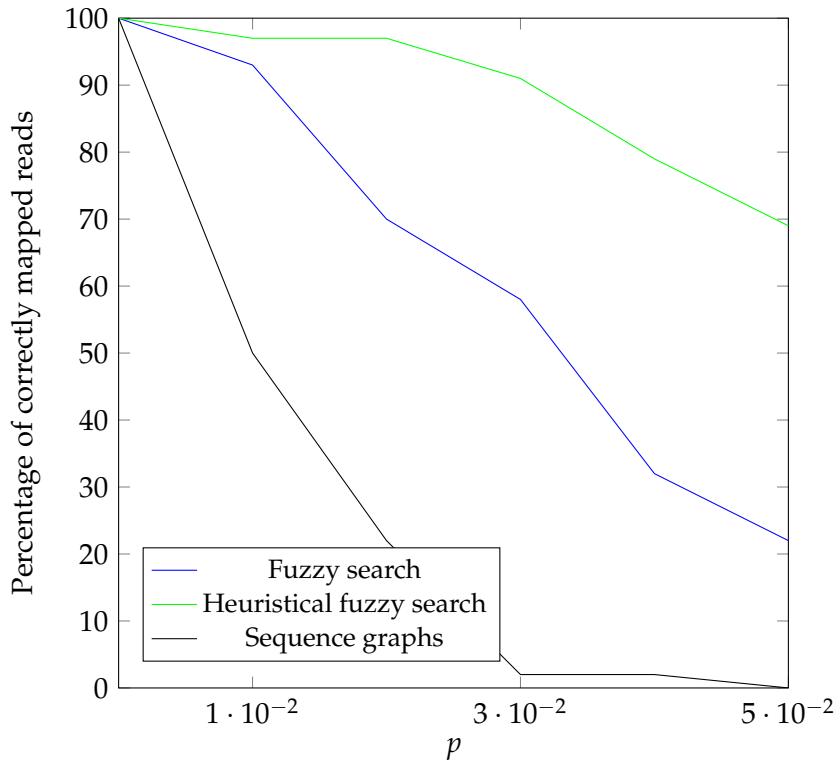


Figure 7.11: Correctly mapped reads from the two tools with $\lambda = 2$

Chapter 8

Discussion

8.1 Is the approach correct?

We will start by discussing the results from Chapter 6. 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.

8.2 Is the approach efficient?

We will move on to discuss the results found in Chapter 7. 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 fuzziness and the indexation. The three subsequent chapters 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 8.4 and also to a large degree form the fundament for the 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 fuzziness and vary the remaining variables, typically related to the complexity of the input data. The results can be seen in figures 7.3, 7.6 and 7.7. In all of these situations the approach display good behaviour when compared to PO-MSA, especially when the runtime is seen as a function of graph size. This in itself is not an extremely impressive result: It can be achieved through simpler solutions, for instance by using hashed k-mers 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 creates a good foundation by showing that 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 fuzziness through the error margin parameter λ . The results can be seen in figure 7.4, where the large exponential growth of the approach can be seen. This is in a way expected: Fuzzyness has the innate property of exponentially increasing the number of interesting probabilities. When we combine this with a dense search space of similar sequences the growth explodes. 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 seem to make the feasibility of the approach plummet. In figure 7.5 there are results which can refute this statement to some degree. We can see that a simple parallelization seem to flatten the growth. There still exists room for large optimizations through parallelizing the suffix tree search¹; we included a simple version to

¹These possibilities are further discussed in chapter 9

display the powerful effect. However, the most important point seen in the figure is the one which was also made while presenting the results: Although the growth is bad, the starting point is decided by the “base case”, which we know will improve as the complexity of the input data increases. The fact that these two factors does not interact with each other² 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 exists a large number of pure combinations of paths of length $|r|$ in the graph, which decreases the probability of not finding anything relevant when doing alignments and similarly decreases the necessity for a high error margin. These two pieces of information leave us arguing that the approach is still an interesting step along the way towards solving the problem.

Indexation

The possibly most visually outstanding result presented in the previous chapter is the pure amount of time used by the indexation process, as seen in figures 7.2d and 7.10. Most of this time is used by slow interactions with the file system, 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[ref], better serialization[ref] and smarter interactions with slower hardware[ref]. Ultimately one could argue that in a real life scenario the index should be kept in memory on a supercomputer[ref][ref]. 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 (see Chapter 9).

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

In the previous chapter we deemed the results from the comparison inconclusive and hard to decipher. Therefore 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 detailed picture when doing comparisons. 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

²This can be deduced by comparing figure 7.4 and 7.5 which uses the same vertical scale

clear design choice taken 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 vertex 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 in describing 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*. Their 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 an 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 fuzziness. 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 does not have a good counter-part in the graph. Sg would see a decrease in quality of the results while we see a growth in runtime. Both can argue these are cases they are not meant specifically to handle. **Transition**.

8.4 Heuristical applications

We first presented the possibilities for finding heuristical alignment which showed themselves through our algorithm in section 4.4. We described our exploration of these possibilites through the `--heuristical=true` parameter in the GraphGenome tool in section 5.2. For reasons previously discussed we moved away from the realm of heuristics through the chapters concerned with testing, except for a brief comparison with the regular algorithm related to accuracy. We will now return to once again considering the viability of this modified approach. The motivation behind this discussion becomes obvious when examining the results: The largest drawback with the approach is the explosive growth in complexity in relation to introduced fuzzyness. In figure 7.8 we can see that when the algorithm is run with the `--heuristical=true` parameter it does not show the same directly linear correlation between the amount of noise and λ . If we can avoid a high error margin, we can avoid the problems brought on by the exponential growth. Showing the validity of the heuristical approach is thus equivalent with significantly increasing the tractability of the approach.

Introducing heuristics is usually done to improve computational tractability, often at the cost of ambiguity: You can no longer assure the found result is correct. The approach we propose in this thesis will only run heuristically when no optimal alignment is found within the constraints of the regular algorithm. This means that for every input triplet G, s, T , where there actually exists an optimal alignment A with a score higher than T , the algorithm will run non-heuristically and thus guarantee optimality. We can even show this holds for results which has a score of $T - 1$: If there existed an alignment with score T we would have found it. There can be no alignment scoring better than $T - 1$ and worse than T^3 . Thus what we have found must be optimal. Whenever we move beyond $T - 1$ we can no longer say anything about the optimality of the achieved result, but we can to some degree guarantee it is a product of smaller optimal solutions. More verbally this can be formulated as *If there exists a good enough alignment the algorithm will find it, if not the algorithm will make an educated guess.* There is even an option of defining what is “good enough” through the input varialbe λ . We claim this is a level of precision attractive to any heuristical algorithm.

8.5 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. The performance results were more ambiguous, revealing that

³Assuming integer scores

the algorithm has both strengths and weaknesses. We have discussed both of these in this chapter, while also proposing possible solutions to some of the arising problems. The fact that there still are shortcomings leads us to conclude that our approach can not serve as an optimal solution to the alignment problem. However, because of the feasibility of the heuristically modified algorithm, we do argue it is an important step along the way to 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 what we assume to be necessary work to utilize we have developed as a stepping stone for finding a better solution to the alignment problem when the reference genome is modelled as 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 be need to be introduced, such as the possibility of searching for reverse complementary strings and knowing the origin of the input data. We consider these domain-specific and trivial to implement. Then we face the problem of our index quickly growing out of control. This is a more intricate problem to solve, but a necessary one to make the approach feasible for real-life datasets. A correlated problem emerges from the exponential growth in time complexity. This is a problem which can be tackled on several levels of technical detail: The most general solution would be to build a more efficient index. Another, more conceptually tractable, solution could be to incorporate restrictions on the input data to avoid the most complex searches. The heuristical algorithm we propose has the inherent property of doing the latter whenever it is faced with problem instances which cannot be solved non-heuristically. Utilizing the knowledge that some results are heuristical could for instance be of importance in an assembly process.

Our heuristical algorithm strictly kicks in whenever the error margin is too low. **This is a due to a “flaw” in our approach, not necessarily general to the problem itself.** One could however imagine situations where the data is simply so large and complex it is impossible to search through it all. We envision this theoretical case such that no amount of optimization would make a full search tractable. In this case we propose that instead of improving the search one builds in a self-validation of the system, a way of scoring the heuristically found alignments. This can be a far less computationally complex operation based on existing scoring routines for regular strings, for instance based around scoring the separate entities making up the alignment. We imagine any such information could be of great importance for an assembly process.

Due to the generality of the problem we defined, the testing done in this thesis has been done on strictly generic data. It would be interesting to, preferably after implementing the improvements mentioned in the first section, test the approach on real genetic data. Determining the value of the flexibility of graph based approaches should be done by comparing it to the regular, linear existing approaches. We boldly claim this would present another dimension of both precision and analytical opportunities.

Bibliography

- [1] Kenneth A. Berman and Jerome L. Paul. *Algorithms: Sequential, Parallel and distributed*. Thomson/Course Technology, 2005.
- [2] 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>.
- [3] 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>.
- [4] 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.
- [5] 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>.
- [6] 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>.
- [7] 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>.
- [8] *Graphviz*. URL: <http://www.graphviz.org/>.
- [9] *GRC Home*. Genome Reference Consortium. URL: <http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/>.
- [10] *GRCh38*. Genome Reference Consortium. URL: <http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/>.
- [11] 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>.
- [12] 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>.

- [13] 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>.
- [14] 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>.
- [15] Arthur M. Lesk. *Introduction to Bioinformatics*. Oxford University Press, 2014.
- [16] Artur M. Lesk. *Introduction to genomics*. Oxford University Press, 2012.
- [17] 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>.
- [18] 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/>.
- [19] Christopher D. Manning, Prabhakar Raghavam and Hindrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [20] 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>.
- [21] 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>.
- [22] 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.

- [23] National Center for Biotechnology Information. National center for Biotechnology Information. URL: <http://www.ncbi.nlm.nih.gov/>.
- [24] 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.
- [25] Adam Novak. *Sequence graphs*. URL: <https://hub.docker.com/r/adamnovak/sequence-graphs/>.
- [26] A. Novak et al. ‘Canonical, Stable, General Mapping using Context Schemes’. In: *ArXiv e-prints* (Jan. 2015). arXiv: 1501.04128 [q-bio.GN].
- [27] 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].
- [28] 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>.
- [29] 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).
- [30] 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>.
- [31] 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>.
- [32] 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>.
- [33] Michael Sipser. *Introduction to the Theory of Computation*. CENGAGE Learning, 2013.
- [34] 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>.
- [35] *The JUnit Framework*. JUnit. URL: <http://junit.org/junit4/>.

- [36] *The Serialization interface: Java API documentation*. Java. URL: <https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>.
- [37] 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>.
- [38] *Understanding the birthday paradox*. BetterExplained. URL: <http://betterexplained.com/articles/understanding-the-birthday-paradox>.
- [39] *Variation graphs*. vgteam. URL: <https://github.com/vgteam/vg>.
- [40] 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>.
- [41] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Pearson Education, 2007.

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 $\max(s) = \text{align}(s, s)$. We also know $\max(s + c) = \max(s) + \text{mappingScore}(c, c)$ 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 exhaustive 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 vertex v_x in every candidate set V'_i , $0 \leq i < |s|$, we need to find the distance to every preceding vertex in every preceding candidate set V'_{i-j} for $1 \leq j \leq \lambda + 1$. We let this number be denoted by $dist(G, s)$

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

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 interchangeable. This is not an entirely true assumption in the cases of shorter contexts at the beginning and end of s , but the impact of 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

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

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 fuzziness and are thus looking for a single context

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

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

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

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

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

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 $|G|$

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

which means

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

and

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

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

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

which can be shortened to

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

Appendix C

The GraphGenome tool

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:

$$B'(n) = \frac{365 - 0}{365} * \frac{365 - 1}{365} * \frac{365 - 2}{365} * \dots * \frac{365 - (n - 2)}{365} * \frac{365 - (n - 1)}{365} \quad (\text{D.1})$$

which can be approximated using taylor series [38]:

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

which gives:

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

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 $\{A, C, G, T\}$.

$$y = 4^{|c|} \quad (\text{D.4})$$

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 vertex 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 vertex 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.

$$x = |G| \quad (\text{D.5})$$

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

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

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 logarithmically approaching 0 (See fig. 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 too 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.

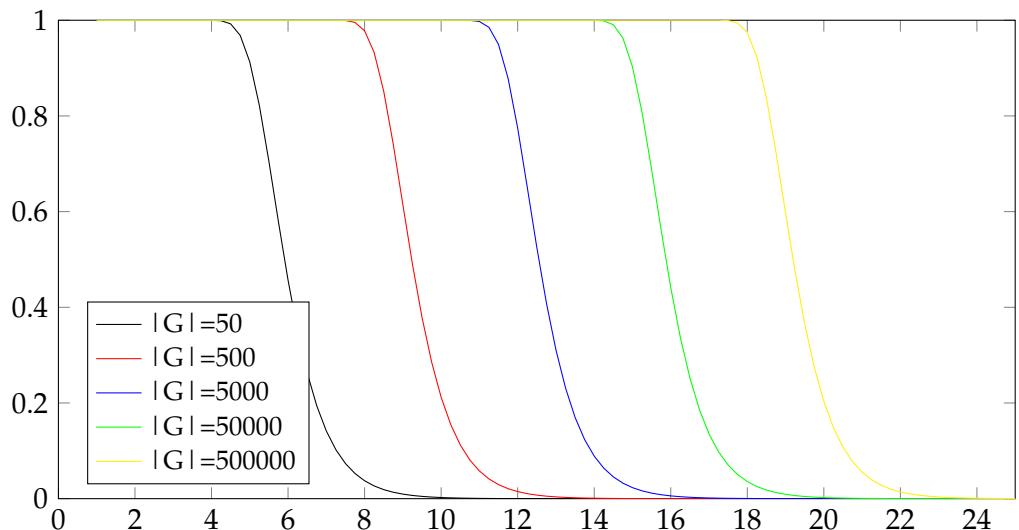


Figure D.1: The functions $y = B(|G|)$ provided by varying $x = |c|$ for different graph sizes