UiO : Department of Informatics
University of Oslo

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Esten Høyland Leonardsen

4th April 2016

Abstract

Due to advancements in DNA sequencing technologies the amount of genetic data has exploded over the last decade. Traditional models for representing said data can not account for the observed variation and more advanced representations are necessary to more accurately depict the true nature of genetical information. However, more complex models calls for more complex techniques for interacting with the data. In this thesis we present an efficient, non-heuristical approach for finding optimal alignments of genetic sequences against graph-based reference genomes.

Acknowledgements

Contents

List of Figures

List of Tables

List of Theorems

Chapter 1

Introduction

1.1 Aims of the thesis

1.2 Contents

1.3 Motivation

Chapter 2

Background

This chapter is divided into three sections. The first is concerned with the biological entities involved throughout the thesis. Because genetics is a huge discipline this chapter will only briefly describe the most critical areas, readers interested in a more thorough introduction are referred to the book “Introduction to genomics”[14]. The second section is directly aimed at the progress in the field of graph-based genomes through discussing relevant articles. Lastly some more general topics of computer science and bioinformatics which play a vital role in the proposed algorithm is presented.

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, which it can bind to to form a *base pair*. 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 completment. Because either of the strands are easily deduced from the other, DNA is usually represented by only of them. DNA can thus be seen as a linear sequence of discrete units and can be represented by text strings, containing the four leading letters representing 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 lookep up in a table called *The standard genetic code*[14, 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[14, Chapter 4], the rest used to be referred to as *junk DNA*. We now know that the non-coding regions also holds important information.

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 [6], 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

this 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*.

2.1.3 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 a dictionary for the genome of a species and can be used in sequencing (Section 2.2.1).

2.1.4 The human genome

The human genome consists of roughly 3 billion base pairs (bp). These base pairs are spread over 46 chromosomes and is assumed to contain about 23 000 genes [14]. The current human reference genome is GRCh38[8], developed and maintained by the *Genome Reference Consortium* [7]. 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 [4].

Major Histocompatibility Complex

The *Major Histocompatibility Complex* (MHC) is a genetic region spanning approximately 4.5-5 million base pairs (mb)[5][20]. 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 [28]. Even more so MHC is known to be a highly variable region, containing variants that are directly associated with disease [9]. The high variability creates difficulties when comparing DNA sequences to determine genetic causes for the observed disorders.

2.2 Bioinformatics

2.2.1 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[24], 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[23] or de Bruijn graphs (Section 2.3.1) are often used in what is known as *de novo assembly*[14, Chapter 1, p. 19].

The different sequencing technologies have varying degrees of errors introduced in their reads, often closely related to the sequencing cost[24]. 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 [31] and correct the reads[16] Can probably provide more citations.

2.2.2 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* (Fig. **??**).

Definition 1 (Mapping score)
A score retrieved by mapping to characters c_1, c_2 *against a substitution matrix. Referenced by* $mappingScore(c_1, c_2)$

The alignment prodecure 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 string does not have a counterpart in the opposing string (Fig. 2.1). When a gap occurs the resulting is penalized according to the length of the gap, by a *gap penalty*.

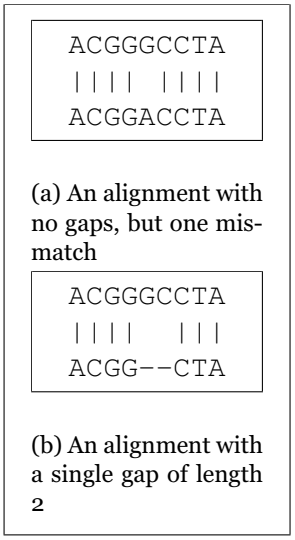
Definition 2 (Gap penalty)
The penalty recieved for a gap of a given length l. *Referenced by* $gapPenalty(l)$.

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. A schema which provides functionality for mapping bases and penalizing gaps is called a *scoring schema*.

Definition 3 (Scoring schema)
A structure which provides a $mappingScore(c_1, c_2)$*-function and a* $gapPenalty(distance)$*-function. The alphabet* Σ *of a scoring schema is defined by the characters present in the scoring schema.*

A gap refers to an element in one of the strings which has no counterpart in the other string when aligned (Fig. 2.1). The scoring schemas

can be based around simple match/mismatch scores, which corresponds to the mathematical *Edit distance problem*, or more complex scores (Fig. 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 another. The two are traditionally solved respectively by the Needleman-Wunsch and Smith-Waterman algorithms which both are based on *dynamic programming* (Section 2.2.3).



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

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

Figure 2.1: Examples of aligned text strings

If more than two sequences are aligned the result is a *Multiple sequence alignment* (MSA). This is typically done on sequences which is expected to share a common ancestor 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.

	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

2.2.3 Dynamic programming

Dynamic programming (DP) is a problem-solving technique where a problem instance is solved by combining the results of smaller subproblems. DP is similar to recursion in that every instance is solved by a *recurrence relation* (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 reoccuring 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 which utilizes a 2-dimensional array to store the computed values (Fig. 2.2).For two strings S and P, every index $[i, j]$ in the edit distance table represents the problem instance of the strings $S[0 : i], P[0 : j]$. The base cases can be seen in the first row and column. There 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:

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

Table 2.2: The 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)

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

where $score(x, y)$ is an inverse equality function. The score for the entire problem instance can be found in the cell with the highest indexes in the

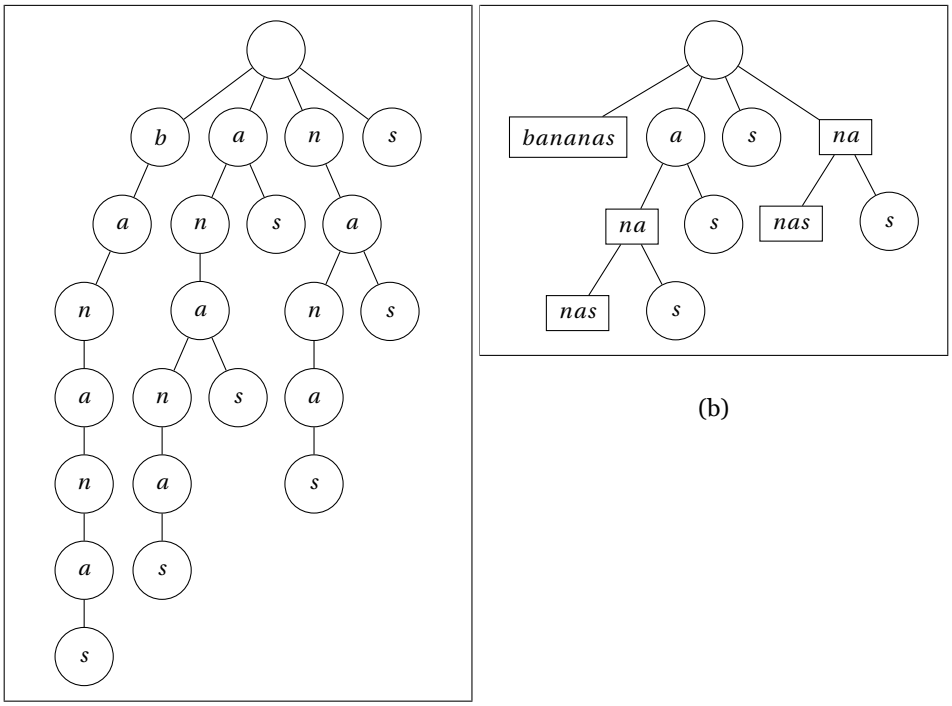
bottom right corner.

There are two separate ways of using Dynamic Programming. A *bottom-up* approach starts at the smallest cases and computes everything until it reaches the actual given problem instances. 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.

2.2.4 Suffix trees

A *suffix trie* is a special tree constructed for specifically for strings of text, containing vertices representing characters (Fig. 2.2a). Every suffix of a string has a corresponding leaf vertice, where the vertices along path from the root to the vertice contains the characters in the suffix. From this it follows that every substring has a matching path starting in the root node. A *suffix tree*, or *compressed suffix trie*, is a suffix trie in which every linear path is compressed into a single vertice (Fig. 2.2b). Suffix trees can easily be extended to hold collections of strings[1, Chapter 20]. An elementary solution 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]

A Block-sorting Lossless Data Compression Algorithm



(a)

(b)

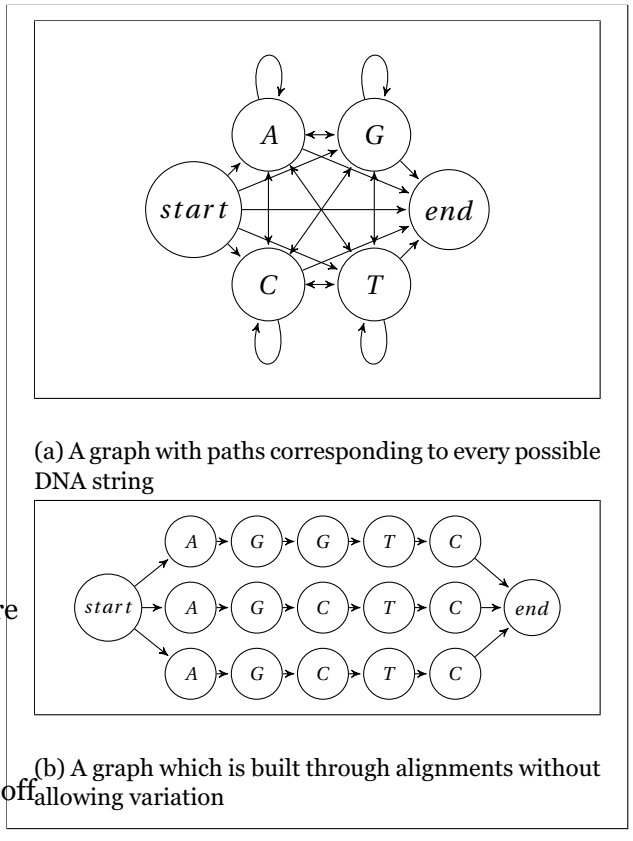
Figure 2.2: The suffix tree (a) and suffix trie (b) of the string “bananas”

2.3 Graph-based genome representations

Representing genetic information as graphs instead of the traditional linear representations have some major advantages. Graphs are far more expressive structures compared to text strings, able to represent more complex relationships between the elements involved. Secondly, if biological questions can be rephrased to graph theoretical settings, the extensive mathematical field of graph theory can present more feasible approaches to previously hard problems. There is however a major problem: A more complex structure 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. This section will present both of these approaches alongside some of the remaining unsolved problems. The section is presented in a way which should not require any previous knowledge of graph theory except for elementary terms, but readers interested in a more complete introduction is referred to the bibliography [27, Chapter 0] [32, Chapter 9] [1, Chapter 11]. Complexity in regards to the graphs and their operations is discussed using *big-O* notation [32, Chapter 2][1, Section 3.1]

2.3.1 Representation

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. Because the input data has variation, the structure needs to tolerate flexibility. There is however a risk of making the structures so flexible they present no consistency, and a flexibility/rigidness-tradeoff becomes apparent (Fig. 2.3). How the struc-



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

(b) A graph which is built through alignments without allowing variation

Figure 2.3: Two sequence graphs displaying too much flexibility (a) and (arguably) too much rigidity (b)

tures are defined in detailed should be determined through the operations which are desirable to perform on them.

De Bruijn graphs

In the article "An Eulerian path approach to DNA fragment assembly"[23], Pevzner, Tang and Waterman proposes *de Bruijn* graphs as a solution to find 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 (Fig. 2.4). 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 lets the problem be formulated 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 vertice representing it.

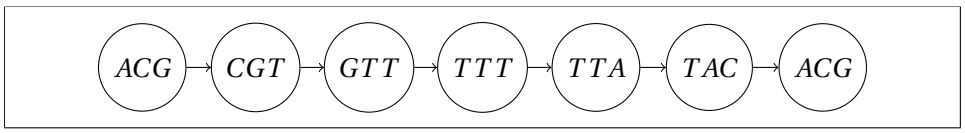


Figure 2.4: A de Bruijn graph with $k = 3$ corresponding to the sequence ACGTTTACG

A more detailed type of de Bruijn graphs is the colored variant where the origins of edges and vertices are stored 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, compared to traditional assembly techniques[10].

Sequence graphs

The relationship between a de Bruijn graph and the sequences it represents is not immediately apparent. A more intuitively pleasing representation is a graph where every vertice contains exactly one nucleotide (Figure figure), a concept called *partially ordered graphs* by Lee et al.[5] and *sequence graphs* by Paten et al.[21]. 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 vertice can no longer be identified solely by the data it contains. To solve this problem the vertices can be given ids, for instance UUIDs[21], for uniqueness. Even though these ids can be used to identify a vertice they contain no information regarding the data which is stored in the graph. 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" 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 adjacendies between the strings in any of the genomes used as input (Fig. figure). 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 vertice is repeated, has at most one vertice 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. If the strictest possible restriction is set, a restriction which requires equal strings, the vertices would contain an exact k-mer from an input sequence just like the de Bruijn vertices.

2.3.2 Mapping

Although the two terms are often used isomorphically we will in this thesis define mapping and alignment as two separate concepts. Mapping is 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. For linear strings mapping is easy. Every string has the same underlying coordinate system, represented by the positions of the characters, and two elements from two separate sequences are either in the same position or they are not. If they are not the difference in position can be derived from the difference between the indexes. Because the indexes of a graph has only one property, uniqueness, they do not hold the intrinsic value of describing relationships 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 relations 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.[21] introduce the concept of *context-based mapping* as a solution to the mapping problem when the reference is modeled as a graph. Context-based mapping is an approach where a vertice is identified by the surrounding environment in the graph. More technically a vertice 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 surrondings. The former defines separate contexts for incoming and outcoming 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 which means every possible context either maps unambiguously to a single vertice or does not map at all. Because the graphs have the possibility of branching a vertice can have several contexts contained in *context sets*. Because every context is unique a collection of such will also be unique, which means context-based mapping leads to a unique two-way mapping schema. This is an even stronger notion of mappability than positions is 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 vertice does not have a unique context it is no longer mappable.

2.3.3 Alignment

As previously discussed, alignment of text strings has for some time been considered a solved problem. 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 one-dimensional representation is no longer sufficient. There does not exist a simple “3 steps before” or “11 steps after” relationship between the elements involved. A solution to this problem can be to imagine alignments agains 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" proposes a direct adaptation of the regular two-dimensional dynamic programming solution for graphs. Every vertice contains a one-dimensional array representing the string which is aligned. Just like an array-index in the edit distance problems, the vertice looks at smaller subproblems to decide what the values of the array should be. However, because this is a graph and not a string, we have no preceding indexes to look up. The vertice has to look at every preceding vertice 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 vertice v in this fashion, the array represents the highest score possible for the substring up to index s for all paths up to vertice 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 susceptible to the inherent complexity of graphs.

Context-based alignment

In the article "Canonical, Stable, General Mapping using Context Schemes"[20] the concept of context-based mapping is used for aligning strings. The algorithm works by identifying substrings 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 to 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* where the α refers to the level of uniqueness required in a context to be considered unique and the β refers to the degree of similarity between a substring and a context in order to classifiy the two as equal.

2.4 Tools and frameworks

2.4.1 Graph representations

2.4.2 Generics in java

2.4.3 Visualizing graphs: The dot-format

Chapter 3

The algorithm “Fuzzy context-based search”

The main concern of this chapter is to introduce the algorithm “Fuzzy context-based search” 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 aswell 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 an 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, and the latter contain exact details and concrete examples.

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.

3.1 The graphs

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

Definition 4 (Graph-based reference genome (Graph))
A pair $G = \{V, E\}$ *where* V *is a set of vertices and* E *is a set of edges.* $|G|$ *denotes the number of vertices in* G.

The involved graphs will be sequence graphs (Section **??**) where every vertice correspond to a single nucleotide from a one or more input sequences used in building the graph. Whether the vertice originates from a single or several sequences is based on whether any new bases has been mapped, and consecutively merged into the vertice. In addition to the nucleobase the vertices will contain an index which is unique to the graph. Every graph G will have two special vertices $s_G = \{s, 0\}$ and $t_G = \{e, -1\}$ which represents unique start and end vertices.

Definition 5 (Graph genome vertice (Vertice))
A pair $v = \{b, i\}$ where $b \in \{A, C, T, G\}$ and i is a unique index. The vertice at index i is denoted v_i. The notation $b(v_i)$ references the first element in the pair (the nucleotide).

The edges 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 6 (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*. Paths is a a way of capturing the combination of several individual characters into text strings in the domain of our graphs.

Definition 7 (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. The distance $d(P)$ between p_0 and p_{-1} is $l(P) - 2$.

Corollary 1
Every edge e is also a path P with $l(P) = 2$ and $d(P) = 0$.

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

There is no correspondence the other way, meaning there can exist full paths which does not originate from a single input sequence (Fig. 3.1). When aligning regular text strings against eachother the introduction of gaps is a key element. The concept of strings with gaps are translated to graphs through *incomplete paths*.

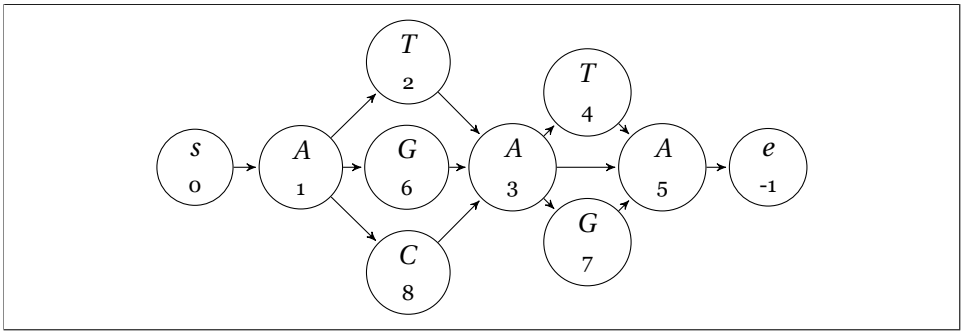


Figure 3.1: An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA". Although the graph is made from 3 sequences, 9 full paths can be found

Definition 9 (Incomplete path)
An 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. We can score an incomplete path by looking solely at the gaps present and avoiding the nucleobases contained in the vertices to produce a *path score*.

Definition 10 (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 summing up the gap penalties for a gap with the distance of the shortest path between every two consecutive vertices.

Corollary 2
$pathScore(P*) = \Sigma^{(}|P* | - 2)_{i=0} gapPenalty(distance(p*_i, p*_{i+1}))$ *where* $distance(x, y)$ *denotes the distance of the shortest path P where $P_0 = x$ and $P_{-1} = y$.*

3.2 The alignment problem

Because the alignment problem is concerned with aligning text strings against graphs we need to define a second component alognside the graphs: The input sequences.

Definition 11 (Input sequence)
A string s over the alphabet $\{A, C, T, G\}$. The length of the string is given by $|s|$. The individual character on position $0 <= x < |s|$ is referenced by s_x

Once we have a clear definition of a graph G and an input sequence s we can try to find an *alignment* between the two, which model the relationship between the two. In order to do this, 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 interal 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-vertice 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 12 (Alignment)
Given a graph G and a string s, an alignment A is an ordered list of length $|s|$ such that every element $a_x \in A$ is either 0 or the index for a valid vertice of G such that for every consecutive pair of valid indexes a_n, a_m there exists a path P where $p_0 = a_n$ and $p_{-1} = a_m$. A 0 represents an unmapped character in s.

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

1. The mapping scores of the mapped elements
2. The gap penalties for gaps in the graph, represented by the distance of the shortest path between consecutive pairs of mapped elements
3. The gap penalties for gaps in the string, represented by unmapped positions

We already know how to find the first two. 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 13
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 $\Sigma_{A* \in A} gapPenalty(|A*|)$ where $A*$ is the maximal unmapped subsequences of A.

Definition 14 (Alignment score)
Given a sequence s, a graph G and an alignment A, the score produced by combining mapping scores for the pairs $\{b(v_{a_x}), s_x\} for 0 <= x < |s|$ 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 $alignmentScore(A)$.

We can then easily define the alignment problem itself:

Definition 15 (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 a highest possible score. This is done in order to simplify the abstract explanations of the algorithm. Implementation-wise this can trivially be changed to find all optimal alignments. The necessary adjustments is discussed as a part of the succeding chapter in section 4.1.3. Additionally we have defined a bounded version of the problem, called *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 16 (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 elements 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.

3.3 "Fuzzy context-based search"

We now present the algorithm we propose as a solution to the bounded optimal alignment score problem. The algorithm consists of two distinct subproblems which are solved in consecutive steps:

1. Create a new 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 has a preset 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. The highest possible score for any string can be found by aligning the string against itself with regular string alignment operations, using the already defined scoring schema. 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 $lambda$ compared to the corresponding element in an optimal alignment can never have a total alignment score higher than T (Appendix A).

3.3.1 Creating a new 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 vertice 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 a higher score than 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 is still a possibility. We formulate a second restriction, to reduce the number of vertices which are not interesting for final alignments:

2. Every vertice $v_x \in V$ 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 vertice 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 only being related to the computational complexity, which means it does not have to be strictly enforced. We can thus relax it:

2. Every vertice $v_x \in V$ should be in V'_i for every $0 <= i < |s|$.

This is a complete relaxation and puts every vertice $v \in V$ in every subset of V'. The resulting parenting candidate vertice set V' is a set far greater than V which is obviously suboptimal for the following search. These two cases represent the two extremes on the scale of how strictly we enforce the second restriction and they both represent problems: Either the search is too hard to the results are not good enough. 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 being the search growing too complex*

We let a vertice v be a *candidate vertice* for index i if it is a part of the *candidate set* V'_i. In order to find candidate vertices we apply *fuzzyness* to the context-based mapping schema proposed by Paten et al.[21]. We say a vertice is a candidate vertice for an index if it has a context which is similar enough to the context of the corresponding position 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 vertice represents the paths passing through it, and because we know that if a context is penalized more than λ compared to the maximal possible score we know that the context can never be a part of a longer incomplete path with a total score higher than T. Thus, more formally, for every index $0 <= i < |s|$ we put v_x in V'_i if and only if the context set $c(v_x)$ of v_x contains a context which can be aligned against $c(s_i)$ with a score higher than T_c. 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 vertice of every candidate set V'_i should have an incoming edge from every vertice 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 vertice looks back for possible paths: Every vertice in every candidate set V'_i should have an incoming edge from *every* preceding candidate set V'_j, where $0 <= j < i$. These edges represent the relationships between the elements of the string. We do also want to represent the relationships between the vertices in the graph they originate from. This is done through introducing *weighted edges*:

Definition 17 (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 value

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 it be inf. At this point we have a complete graph G'.

Corollary 3
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 vertice is connected to every preceding vertice, 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 sets a bound both on how far back in the candidate sets a vertice looks for neighbours and, more importantly, puts a strict upper bound on the complexity of the individual graph searches done in G.

3.3.2 Searching the newly formed graph

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

In order to continue securing optimal results the algorithm does the search using dynamic programming. The search algorithm is conceptually very similar to PO-MSA[12], 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 canditate nodes 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 the scores to be the highest possible score for aligning the substring of s ending in 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 $mappingScore(b(v_x), s_0)$, 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 $-2 * \lambda$, which yields any path starting with the vertice score lower than T and thus not be considered candidates for the optimal alignment.

The recurrence relation of the dynamic programming algorithm is concerned with setting the score for any vertice/index pair which is not a base case. The score for these candidate vertices $v_x \in V'_i$ are set by looking at all incoming edges, find the one yielding the highest score and add $mappingScore(b(v_x), s_y)$. The score for an edge is found by taking the score 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, in all edges traversed in the final alignment will only be penalized for one of them.

When the scores have been computed for every candidate vertice we can start looking for the highest score, which represents the alignment score for one of the optimal alignments. We will find this score in 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.

Chapter 4

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 concrete example which is found in figures 4.1-4.3 and table 4.1. Throughout this chapter, and in the example, the scoring schema is assumed to be the *negated edit distance* schema.

Definition 18 (Negated edit distance scoring schema)
A scoring schema where the substitution matrix is a variant of an identity matrix scoring equal characters 0 and mismatches -1. Both the gap opening and gap extension penalty is -1

The reason for the negation compared to regular edit distance is that the tool is implented 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 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 what readers can expect when seeing the results and using the tool themselves.

4.1 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 theprecomputation is dependant only on the graph and the index is thus reusable for several alignments.

4.1.1 Building the index

There are two data structures needed for aligning a string against the graph: a suffix trie for left contexts and a suffix trie for right contexts. Before either of the two are built the algorithm needs to decide a length for the contexts. Currently 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 (Appendix A) but will have an impact on the runtime (Appendix B).

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

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

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

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

```
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

incoming and an outgoing neighbour is switched and the algorithm starts again to generate right contexts.

As sets per definition does not allow duplicates the impact of a branching occuring in the graph will fade away after exactly $|c|$ steps as the difference is trimmed away (Fig. 4.1), and thus avoid explosive exponentiality in the context set sizes. Unlike Paten et al. [21] 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.

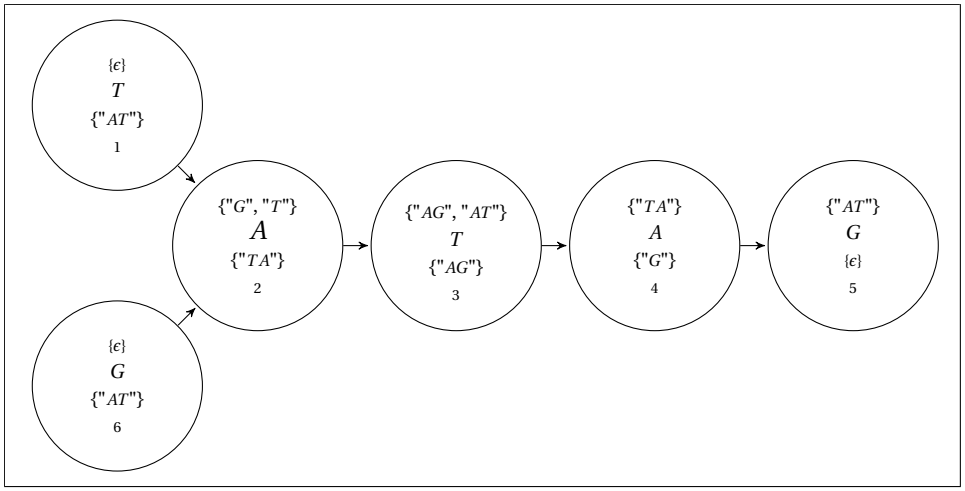


Figure 4.1: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

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. 4.2). In theory every node can have $4^{|c|}$ contexts in each set, in practice a more fair approximation is $b^{|c|}$ where b is the observed branching factor for the graph. Should contain something about probable values of B. Find an article on it. 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, looking up approximately b neighbours. Building the entire index can thus be done in $O(2|G|b + 2|G|b^{|c|}|c|)$. compress more?

4.1.2 Generating the modified graph

Unlike the index built in the previous step, the produced 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 it's corresponding suffix trie, where search is a recursive function based on PO-MSA. The root node is supplied with a one-dimensional scoring array corresponding to the context string c, which is initialized with all zeroes. Then, for every child, a new scoring array 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 node:

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

(Reference actual code in supplementary?), (more explanation needed?). An important aspect is that this procedure uses the same scoring schema as the one defined for the entire alignment. This newly created array is then supplemented to the same recursive function in the child. When a leaf node is reached the last index of the supplied scoring array corresponds to mapping the entire string c against the entire context achieved by concatenating the characters contained in the path through the tree traversed by the recursion. If the score is higher than the context threshold T_c for the given context string, every index contained in the node 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 vertice 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 which we called something smart. 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 node. 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.

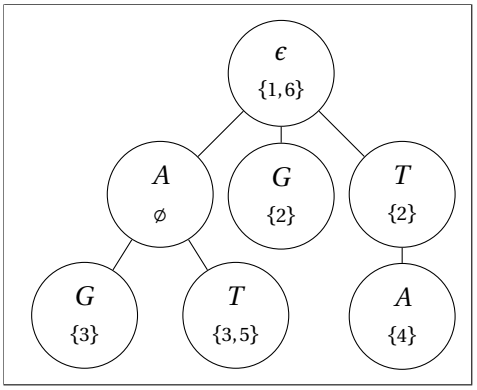
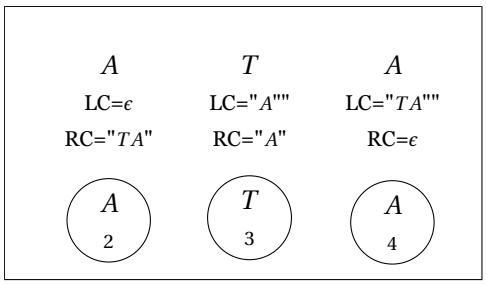


Figure 4.2: The left suffix tree corresponding to the graph in 4.1

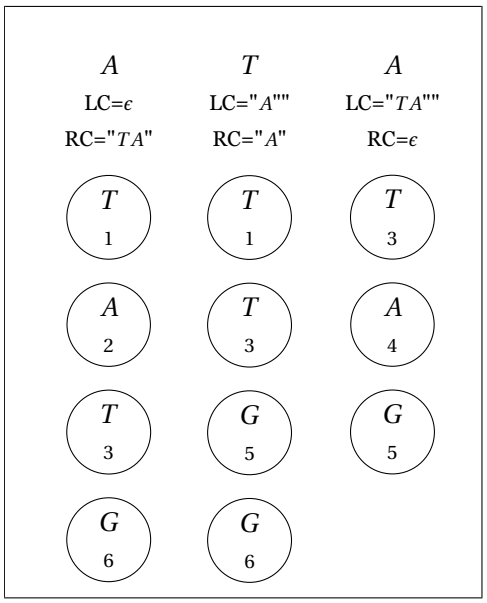
Additionally the suffix search does one more optimization. Whenever the context is short, defined as *shorter than* $|c|$, there will be alot of matches, and the tool has to iterate over every single one to subtract its indexes. In these cases the tool simply handles returns an empty set which is treated 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 node 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 node 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 (something alot smaller. Needs calculations).



(a) $T = 0$



(b) $T = 1$

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

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

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

Intuitively this should be the place where the edges are created in order to finish up G'. As mentioned 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 are 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 point during the computation.

4.1.3 Searching G' with a modified PO-MSA search

The first thing done in the search is to 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 by Strings.

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

(a) Indexes

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

(b) Scores

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

(c) Backpointers

Table 4.1: The 4 arrays used by the searching algorithm when using the candidate sets from Fig 4.3 and $T = -1$

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

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

The iteration over the elements of the set with the counter j will not be according to any ordering as the elements of the set are inherently not ordered. This is not important to us as the elements of a single candidate set have no relation which we want to contain. However, from this point onward, we have given the elements an order. Although this 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 index.

After setting the values in the easily discoverable 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 the indexes $1 \leq i < |s|$ are looped over with j as a counter, and $indexes[i][j]$ is set to x. For every such entry a list of pairs is made with other indexes (i', j') such that i' is a preceding index $i' < i$ and j' is another counter variable looping over $indexes[i']$. For every entry-pair $((i, j), (i', j'))$ we produce a score by the scoring function $\theta((i, j), (i', j'))$. The scoring function works by combining the score contained in the preceding entry, $scores[i'][j']$, the gap penalties, and a mapping score for the current index $mappingScore(v_{indexes[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length

$distance(v_{indexes[i'][j']}, v_{indexes[i][j]})$. The computation of the last gap penalty corresponds to finding the edges, which up to this point have been without interest to the algorithm. We search for these distances by a breadth-first regular graph search which starts in the vertice $(n_{indexes[i'][j']}$ and is concluded when we find $n_{indexes[i][j]}$. Whenever we search for more than λ steps without finding the target vertice, we can return the current distance multiplied by 2. When a gap penalty is computed for a gap this length the score will always be $2 * \lambda$ which means the edge can never be part of any final alignments and is thus not interesting.

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

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

where θ is a scoring function defined as:

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

When the iteration ends we will have a score for every candidate vertice in every candidate set. We can iterate over the scores corresponding to the last candidate set, $scores[|s| - 1]$ to find the highest alignment score. The optimal alignment ends in the vertice with the index in the corresponding cell in the $indexes$ array. We can then backtrack backwards through the $backPointers$ array, storing the corresponding indexes of the vertices from the $indexes$ array along the way to produce the actual alignment. The entire operation can be done in $O(INSERT)$ (Appendix **??**).

Finding all optimal alignments

There is only a small adjusment 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.

4.1.4 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 (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 a 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 unmappable.

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 shemes). 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.

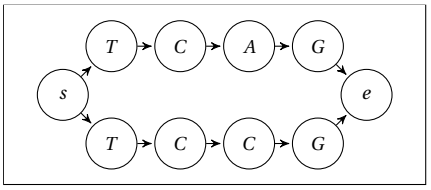
The second case is often less interesting as it usually leads to vastly less coherent alignments and as a result typically lower scores. Even so, there are interesting concepts touched upon also here: Because of the distinct penalty given to invalid edges we have no real way of knowing how the proposed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

4.2 Merging aligned sequences

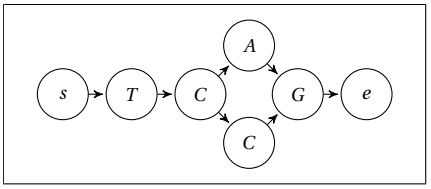
Whenever a graph is made from a set of genetic sequences there is an expectation of what the graph should look like, where there should be branches, common sequences and so on. This is decided by how the input sequences are merged together to produce graphs. Although the merge can be seen as the key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

	T	C	A	G
T	0	-1	-2	-3
C	-1	0	-1	-2
C	-2	-1	-1	-2
G	-3	-2	-2	-1

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



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



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

Figure 4.4: Different scoring thresholds T yields different reference graphs

Whenever an input string s is aligned against a graph G we can do a merge, a procedure which should result in the internal structure of s being present in the merged graph $G*$. To make reasoning around the procedure we visualize s as a special sequence graph G_s where every individual character is represented by a single vertice 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 vertice in G_s with position x. An important note is that this is a simplified type of the previously defined graphs which does not require start and end-vertices, which means the indexation begins at 0. The reason for this simplification is that we can use the alignment A to create an equality operator:

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

When we have this way of checking equality between the two graphs we

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

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

There is one important aspect to be considered when using this 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 shows that even though the tool is designed for aligning short reads against a large graph it should not also be used to merge these in. The result would be a long full path representing a chromosome, genome or similar and several short full paths representing the individual paths. The imagined usage of the tool consists of more steps:

1. Create a graph by merging several known genetic sequences
2. Align individual reads against the graph
3. Combine the reads to a sequence similar to the originating sequences
4. Merge the result of the combination

The tractability of the not implemented combination process is discussed in Section SOMEWHERE

Chapter 5

Experiments

The following chapter describes the details of the experiments conducted to produce the results in the succeeding chapter. The experiments are divided along a natural border, decided by the size of the input data, into two classes. Each class has its own section describing the motivation behind the experiments and the details specific to that class. Elements which are shared among the classes are discussed in once in the last section of the chapter.

5.1 Proof of concept

Whenever a graph is built from a set of sequences one can get an intuition concering what the final result should look like. These experiments are attempts to formalize the notion of intuition into stable, testable results. Due to readability and shortcomings of printed media only a small set of the experiments are presented here. A more exhaustive set of tests can be found as unit tests in the tool (Appendix C).

5.1.1 Test data

Because the motivation behind these tests are to determine the behaviour of the algorithm, the input data consists of small, handcrafted sequences which for each experiment contains exactly one easily identifiable trait. These traits are crafted in a way which reflects the nature of variation in genetic sequences. Because the negated edit distance scoring schema is a flat scoring schema which penalizes all errors the same it is prone to display order of operations characteristics of the underlying algorithm. Because the order of operations of the implementation is well known to the authors this is taken into account when creating the data.

5.1.2 Validation

There are two main concepts which the validation of this experiment class wish to capture: The intuition and the formalization. The intuition is captured through visualizable results. Every experiment will provide a

visualization of both the inputs and the outputs. When the inputs are visualized, one of the sequences will be chosen as a basis for the graph. The output visualizations will be directly produced by the tool using the -print parameter, followed by porting the resulting dot-file to the tikz syntax used in this thesis5.1. The formalization is carried out through a set of statements from first order logic concerning the state of either the resulting graph or the alignment, which are verifyable in the visual outputs. The mentioned unit tests are created to represent these statements through Java syntax.

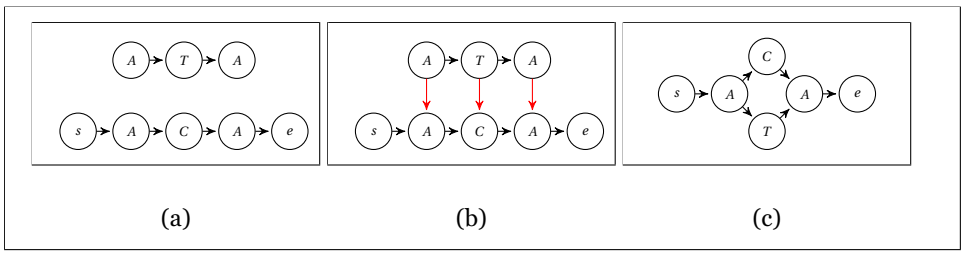


Figure 5.1: The syntax of the visual outputs. Two input sequences “ACA” and “ATA” are given, and the first is chosen as a basis for the reference graph (a). Then the second sequence is mapped against the graph (b) and merged (c)

5.2 Efficiency

In order to determine the usefulness of a new approach it should be compared to other already available approaches. The goal of these experiments is to run the implementation in the tool against similar applications to determine the grade of correctness and the computational feasibility of the approach. Because of its guarantee for optimality PO-MSA (Section 2.3.3) is chosen as a baseline, through an own implementation in Java. The results are compared with the vg implementation[30] and the tool corresponding to the article "Canonical, Stable, General Mapping using Context Schemes"[19].

5.2.1 Test data

Because the experiements are meant to reflect usage in an everyday situation the tests are run on real genetic data fetched from the vg github repo[30], and from the test-set provided forthe previously mentioned tool made by Noval et al. The sequences correspond to allelles of the MHC region2.1.4 and chromosome 6 in the human genome. All the data used can be found in the github of the "Fuzzy context-based search" tool (Appendix C).

In order to do an alignment there needs to exists reads aswell as the reference graph itself. A read for a graph G is generated by the following procedure:

1. Choose a read length l, an SNP-probability p_s a deletion probability p_d and an insertion probability p_i
2. Choose a random vertice $v_x \in G$ where every path to the end vertice t_G has a length $\geq l$
3. For r steps:
   (a) Append $b(v_x)$ to the read r
   (b) Set the new v_x as a random neighbour of the old v_x
4. Add noise to r according to p_s, p_i and p_d.

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 2.2.1). why is this ok. In order to provide reproducability the randomness in the reads are generated from a seed.

5.2.2 Validation

When an alignment is produced for a read it is classified either as optimal or not optimal. Intuitively this can be determined by whether the generated read aligns back to the path it was generated from. However, when noise

is introduced an interesting phenomenon can occur: The modified read can be more similar to another path than its origin. This can also occur whenever there exists actual equal paths in the graph, typically in the case of repeats. In order to stick with mathemathical properties, 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 approaches produce different alignments than PO-MSA these can be classified as optimal. This falls within the problem definition (Definition 16).

5.3 Common elements

5.3.1 Scoring schema

5.3.2 Hardware/runtime environment

Briefly about hardware/environment

There is one important aspect of the efficiency experiments which should be taken into consideration: The PO-MSA and "Fuzzy context-based search" algorithms are both own implementations, which means the time capturing mechanisms are wrapped closely around the algorithms themselves to rule out unrelated noise. This could lead to unfair overhead with regards to the running times of two other tools which are tested. Even though the data sets provided should be large enough to minimize this overhead and depict the true functions underlying the algorithms this is a consideration which is taken when discussing the results.

Chapter 6

Results

6.1 Proof of concept

This subsection can be an introduction to the usage of the tool as much as it is a display of results. The results will be presented through a series of commands and the visual output produced. In addition to this a set of first order logic formulas regarding the state of the graph is given in the syntax of JUnit tests. As the syntax of the tests are rather intuitive there is no prior knowledge to JUnit necessary.

We start out by building a graph from a single sequence: `>java -jar ../target/graph-genome.jar index -index=equal.index -input-sequences=ACGTATTA -png=ref`

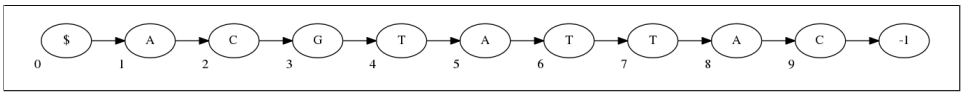


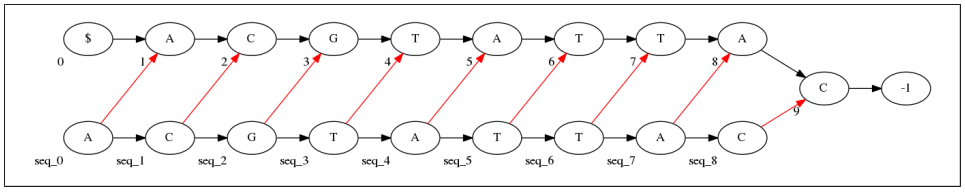
Figure 6.1: A reference graph made from the sequence “ACGTATTAC”

This will be the graph used as a starting point for all the test cases. In order to separate the cases completety and avoid confusion every test uses a different index, denoted by the `-index` parameter. Additionally the `-png` parameter is used in all the cases as this is responsible for producing the graphical results.

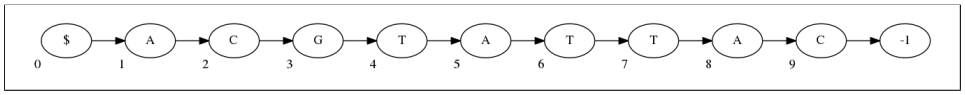
6.1.1 Equal sequences

We start out by aligning an equal sequence. In order to produce both an alignment and the result of merging in the sequence the tool is run twice, once with the `merge` parameter set. `>java -jar ../target/graph-genome.jar align -index=equal.index -input-sequence==ACGTATTA`

```
>java -jar ../target/graph-genome.jar align -index=equal.index
-align-sequence=ACGTATTAC -png=equal-alignment
>java -jar ../target/graph-genome.jar align -index=equal.index
-align-sequence=ACGTATTAC -merge=true -png=equal-merge
```



(a)

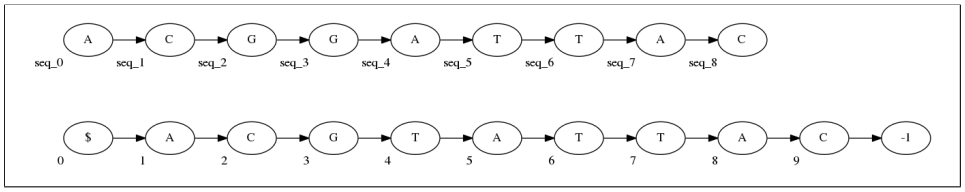


(b)

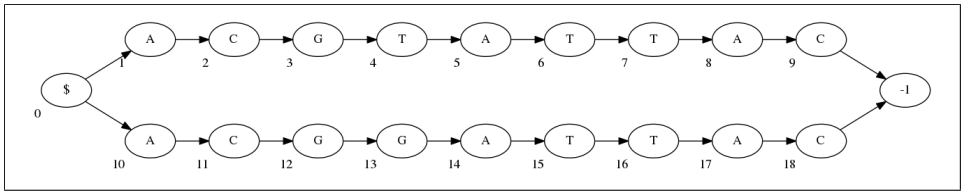
Figure 6.2: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference graph seen in Fig. 6.1

6.1.2 SNPs

```
>java -jar ../target/graph-genome.jar align -index=snp-error.index
-align-sequence=ACGGATTAC -png=snp-no-margin-alignment
>java -jar ../target/graph-genome.jar align -index=snp-error.index
-align-sequence=ACGGATTAC -merge=true -png=snp-no-margin-merge
```



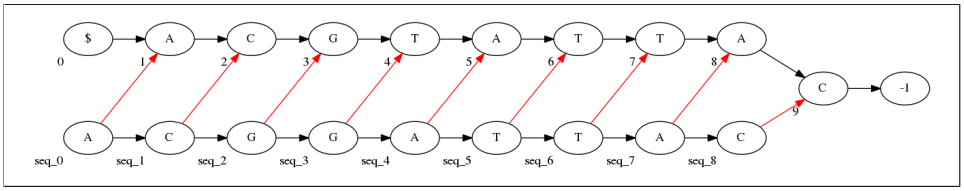
(a)



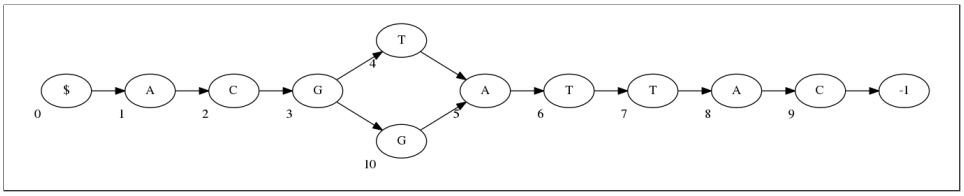
(b)

Figure 6.3: The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference graph seen in Fig. 6.1 with $\lambda = 0$

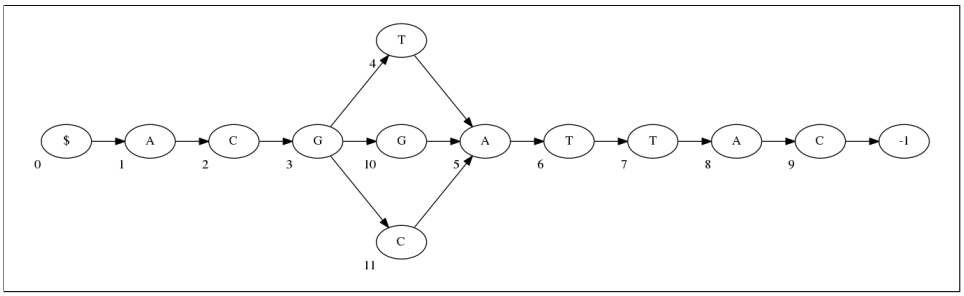
```
   >java -jar ../target/graph-genome.jar align -index=snp.index
-align-sequence=ACGGATTAC -png=snp-alignment -error-margin=1
>java -jar ../target/graph-genome.jar align -index=snp.index
-align-sequence=ACGGATTAC -merge=true -png=snp-merge -error-margin=1
java -jar ../target/graph-genome.jar align -index=snp.index
-align-sequence=ACGCATTAC -merge=true -png=snp-merge-two
-error-margin=1
```



(a)



(b)

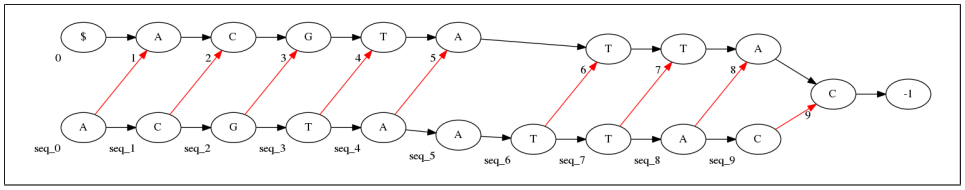


(c)

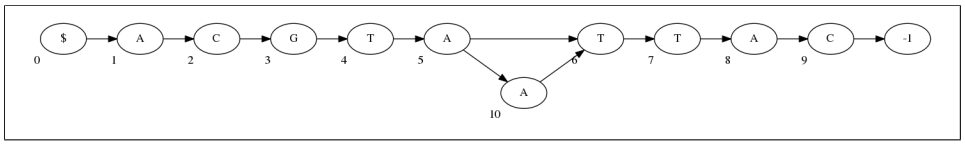
Figure 6.4: The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference graph seen in Fig. 6.1 and then merging the sequence "ACGCATTAC" (c) with $\lambda = 1$

6.1.3 Indels

```
>java -jar ../target/graph-genome.jar align -index=insertion.index
-align-sequence=ACGTAATTAC -png=insertion-alignment -error-margin=1
>java -jar ../target/graph-genome.jar align -index=insertion.index
-align-sequence=ACGTAATTAC -merge=true -png=insertion-merge
-error-margin=1
```



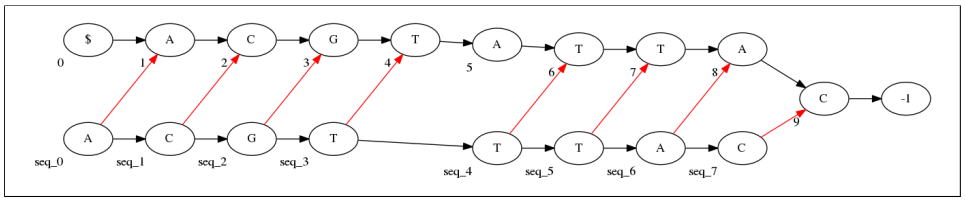
(a)



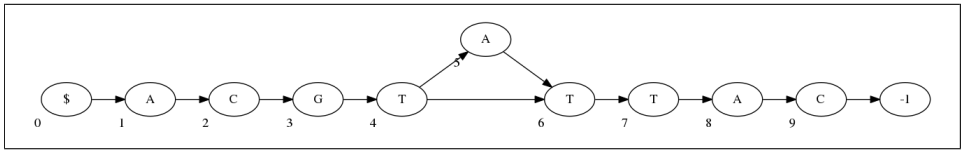
(b)

Figure 6.5: The result of aligning (a) and merging (b) the sequence "ACGTAATTAC" against the reference graph seen in Fig. 6.1

```
  >java -jar ../target/graph-genome.jar align -index=deletion.index
-align-sequence=ACGTTTAC -png=deletion-alignment -error-margin=1
java -jar ../target/graph-genome.jar align -index=deletion.index
-align-sequence=ACGTTTAC -merge=true -png=deletion-merge
-error-margin=1
```



(a)



(b)

Figure 6.6: The result of aligning (a) and merging (b) the sequence "ACGTTTAC" against the reference graph seen in Fig. 6.1

6.1.4 Structural variations

6.2 Efficiency

6.2.1 Building the index

6.2.2 Alignment

Runtime as a function of |G|

Runtime as a function of |s|

Runtime as a function of λ

Chapter 7

Discussion

Chapter 8

Conclusion

Chapter 9

Further work

9.1 Improvements to the algorithm

9.2 Heuristical approaches

Appendices

Appendix A

Proving optimality

Appendix B

Complexity analysis

Appendix C

The GraphGenome tool

Appendix D

The birthday problem

Bibliography

[1] Kenneth A. Berman and Jerome L. Paul. *Algoritms: Sequential, Parallell 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] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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].

[22] 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.

[23] 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).

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

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