UiO : Department of Informatics
University of Oslo

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Esten Høyland Leonardsen

23rd March 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.

Contents

List of Figures

List of Tables

List of Theorems

Preface

Chapter 1

Introduction

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 molecule 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 contain one of the nitrogenous bases *Adenine* (A), *Cytosine* (C), *Guanine* (G) or *Thymine* (T). 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. In the DNA molecule two reverse complementary 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 *complementary base*, A with T and C with G. The paired nucleotides are called *base pairs*. Because either of the strands are easily deduced from the other, DNA is usually represented by only of them. DNA can be seen as a linear sequence of discrete units and can thus 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*. All other 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 same three bases A, C and G aswell as *Uracil* (U). The mRNA can be divided into *codons* which are triplets of nucleotides encoding *amino acids*, which are the building blocks of 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 is often referred to as *junk DNA*.

2.1.2 Variation

Genetic information is prone to mutations, either as a result of environmental influence or as a consequence of imperfections in reproduction. 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 to form a *gene pool*. These different genotypes give rise to a variety of phenotypes. A subset of these phenotypes, and their associated individuals, can be better suited for survival and reproduction than others. Given enough time and enough scarcity in resources the best suited individuals will survive and pass on their genes to the next generation. This is the process of *natural selection* which is the main driving force behind evolution. Another mechanism in play is *genetic drift* which affects gene frequencies in a gene pool through non-selective, random processes.

Because there are more possible combinations of codons than there are amino acids in the standard genetic code there exists some overlap between the nucleotide triplets 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 2.1.5).

2.1.4 The human genome

The human genome consists of roughly 3 billion base pairs (bp). These base pairs are spread over 22 paired 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 million base pairs (mb) NEEDS CITATION. 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].

2.1.5 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 2.2.1) 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.

2.1.6 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*.

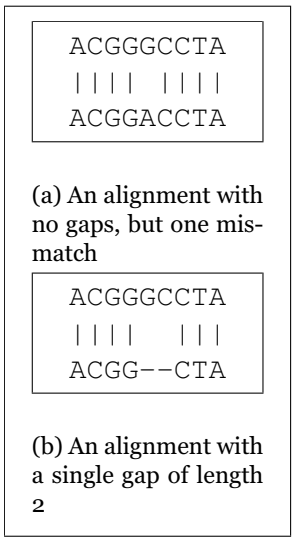
Definition 2 (Gap penalty)
The penalty recieved for a gap of a given length l. *Referenced by* $gapPenalty(l)$.

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 2.3.1).



(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 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. No graph theoretical foreknowledge is needed as all the involved elements will be defined before they are used, but for interested readers there exists good sources in 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]

2.2.1 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. 2.2). How the structures 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. 2.3). 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

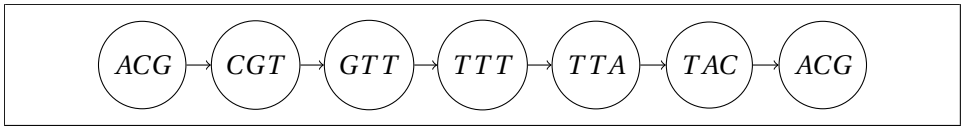
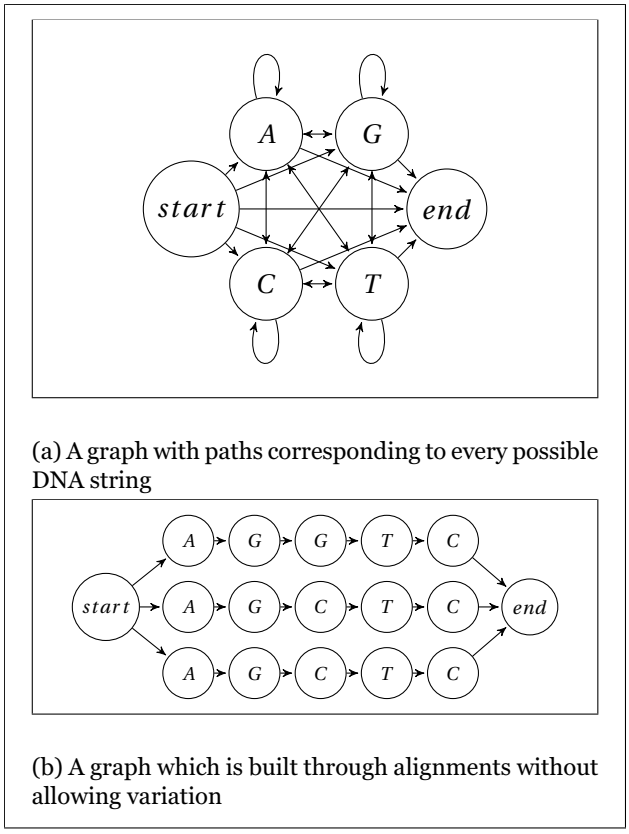


Figure 2.3: A de Bruijn graph with $k = 3$ corresponding to the sequence ACGTTTACG

disambiguity: Any legal k-mer has at no point more than one vertice representing it.

A more detailed type of de Bruijn graphs is the colored variant where the origins of edges and vertices are 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].



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

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

Figure 2.2: Two sequence graphs displaying too much flexibility (a) and (arguably) too much rigidity (b)

Cactus graphs

Cactus graphs for genome comparisons

Sequence graphs

WRONG: POMSA. The term *sequence graph* stems from the article "Mapping to a Reference Genome structure"[21] and describes a graph structure which is possibly more intuitively pleasing. Every vertice in the graph corresponds to a single nucleotide from one or more genetic sequences used in building the graph. An edge represents two nucleotides

which are consecutive in one of the original sequences. Paths symbolize subsequences of the originating input sequences. To handle the arising problem that the contents of nodes are no longer unique each vertice can be given an index which is exclusive for their associated graph. Whenever a graph is referenced without being specifically classified the following definitions are assumed:

2.2.2 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 paths that pass through the 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 it's 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 two-way unique mapping schema. This is an even strong notion of mappability than positions is strings, as a character of a string does not necessarily map uniquely back to its position. This strong notion has a drawback: There exists situations where a vertice does not have a unique context which yields it unmappable.

2.2.3 Alignment

Intro

Dynamic programming on graphs

PO-MSA

Context-based alignment

Canonical, Stable, General Mapping using Context Schemes

2.3 Techniques and tools

2.3.1 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.

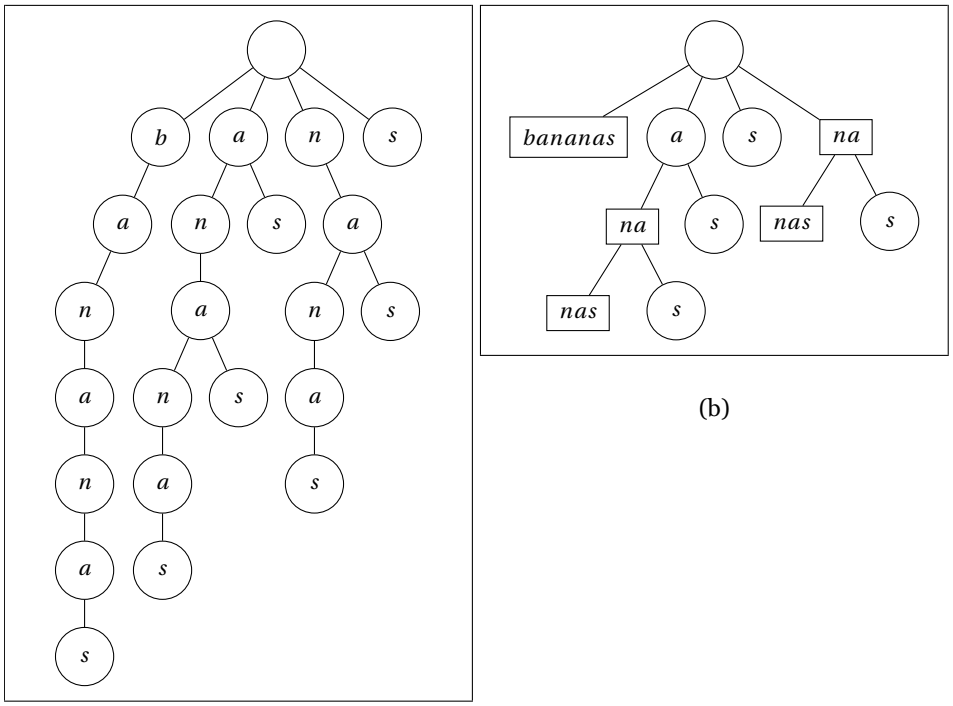
2.3.2 Implementing graphs

2.3.3 Suffix trees

A *suffix trie* is a special tree constructed for specifically for strings of text, containing vertices representing characters (Fig. 2.4a). 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. 2.4b). 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

2.3.4 Visualization of graphs: The dot-format



(a)

(b)

Figure 2.4: The suffix tree (a) and suffix trie (b) of the string “bananas”

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. 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 all the involved elements.

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 2.2.1) where every vertice correspond to a single nucleotide from a one or more input sequences used to build 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 within the surrounding 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 atleast one input sequence.

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 the introduction of gaps is a key element. The concept of strings with gaps are translated to graphs through *incomplete paths*.

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*. 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.

Definition 10 (Path score)
$pathScore(P*) = \Sigma^{(}|P* | - 2)_{i=0} gapPenalty(distance(p*_i, p*_{i+1}))$ *where* $distance(x, y)$ *denotes the distance of the shortest path* p *which starts in* x *and ends in* y.

3.2 The alignment problem

Because the alignment problem is concerned with aligning text strings against graphs we need to define another component: 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

SOMETHING SMOOTH. We are trying to find *alignments* which model the relationships between a graph G and an input sequence s. 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

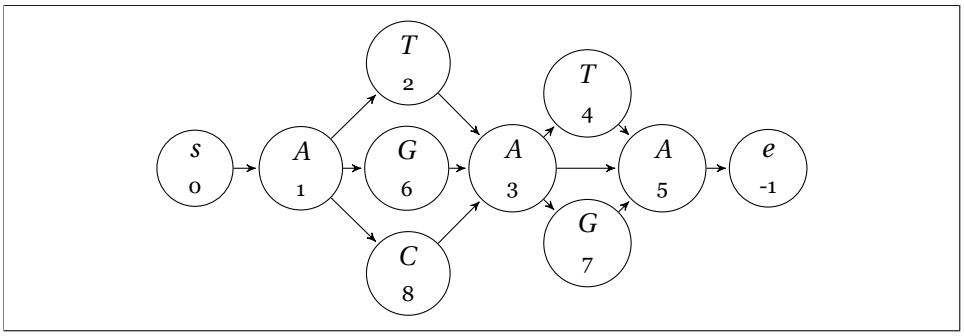


Figure 3.1: An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA". Although the graph is made from 3 sequences, 9 full paths can be found

being mapped to 0 which is 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. Moving forward through the individual positions s_x which are mapped does then correspond to moving through the incomplete path in G.

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, modelled 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 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 $A*_{x:y}$, 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 13 (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 gaps in A.

We can then easily define the alignment problem itself:

Definition 14 (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 explanations of the algorithm and implementations can trivially be changed

to find all optimal alignments. The necessary adjustments is discussed as a part of the Implementation chapter in section 4.1.4. 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 15 (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 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 two-dimensional alignment procedures, using the same scoring schema as for the remainder of the procedure. Introducing λ gives us the opportunity to do strict pruning throughout the entire process of alignment: 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 TA.

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 parts along the horizontal direction of the graphs are interesting for the alignment. The same argument can be made for extremely complex graphs, where only a small number of the branches are relevant, in an operation we have called a *vertical pruning*. The result of both kinds of pruning is a vertice set V' consisting of *candidate vertices*, vertices which are deemed relevant for the final alignment. The candidate vertices are grouped into *candidate sets* V'_x where the set with index x corresponds to the character $s_x \in s$. The conceptual idea is that every vertice which has a possibility of being mapped to the $x - th$ element in the optimal alignment exists in V'_x. Finding out which vertices should be in the candidate sets is done through *fuzzy context-based mapping*. Fuzzy context-based mapping is based on introducing *fuzzyness* to the context-based mapping schema proposed by Paten et al. [21]. The context set of a vertice v_x is denoted $c(v_x)$ and refers to every path which passes through v. The fuzzyness denotes how different the contexts can be and still be considered as viable candidates, the exact amount of fuzzyness is given by λ. The elements of every candidate set can be found by fuzzy searches on efficient tree-structures containing the contexts of the vertices, represented as strings.

When the vertices of the new graph has been found we move on to create the set of edges E'. The goal of the edges in E' is to provide a relation between two vertices $v'_x, v'_y \in V'$ which can be used in finding an optimal alignment. Explicitly there are two relationships which should be modelled: The relationship between the vertices in the original graph G and the relationship between the positions of the candidate sets containing the vertices. As the candidate sets are directly connected to the indexes of s they follow a natural ordering where distances can be easily measured. The relationship between the two vertices in the graph is the distance $d(P$ of shortest path P which starts in v_x and ends in v_y. In order to incorporate this information into our model we introduce *weighted edges*.

Definition 16 (Graph genome weighted edge (Weighted edge))
A triplet $e' = i_s, i_e, w$ where the two first elements are indexes for vertices in the graph G' and the last element is an integer value.

Merging both these relationships would seem to present some conceptual problems. Firstly, whenever considering non-linear gap penalties the two should be considered as separate distances. Secondly, allowing "double gaps", aligned gaps in the input sequence and the path in the graph, should not be allowed in the optimal alignment. This is partially handled by the searching step of the algorithm: Because the distances, and thus the penalties, are combined the algorithm will always prefer a path going through a vertice as this only introduces a single gap penalty. The remaining cases all involve gaps which are not prefixing or suffixing the alignment, i.e. gaps where there is a vertice v_s and before the gap and a vertice v_e after the gap. Because the algorithm always choses a path through a vertice when it can,

this means there are no viable vertices between v_s and v_e chosen as candidates. Viable in this case refers to every vertice on every path starting in v_s ending in v_e. Because there are no candidates, every context for every vertice on every path between the two is penalized more than λ in regards to an optimal alignment. This means no incomplete path which traverses these paths can ever possibly score higher than T, and the algorithm recognizes the path as not interesting.

In order to represent the transitions between the positions of the text strings every vertice in every candidate set should be connected to every vertice

3.3.2 Searching the newly formed graph

Chapter 4

Design

In this section we will present the algorithm “Fuzzy context-based search” as an approach for aligning text strings against graph-based reference genomes. First the remaining elements involved in precisely defining the problem will be described. Then the syntax of the reference graphs used are explained in more detail. Finally, the algorithm is presented both through a conceptual overview and in specific detail. The implementation details refers to the *GraphGenome* tool (Appendix C).

4.1 Aligning sequences

4.1.1 Overview

"Fuzzy context-based search" is the algorithm we propose as a solution to the bounded optimal alignment score problem. The first step of the process is to build a searchable index based on the given graph G. This index is independent from the input sequences to be aligned, and can thus be reused for several searches. The alignment itself consists of two steps: Building a new graph G', from both G and an input string s, and search this newly formed graph for the optimal alignment. Searching for an alignment means combining nodes, representing bases, into a path which represents a linear sequence. This linear sequence can be aligned against the input sequence with regular string alignment tools and is therefore easily scorable. If the algorithm finds an alignment this is guaranteed to be one of the paths which produces the highest possible alignment score for any path in the graph (Appendix A). There are some situations where the algorithm results in an empty alignment. These cases will occur when there are no paths in the graph which produces an alignment score higher than the defined threshold T, and the sequence s is identified as unalignable. When an empty alignment is provided as a basis for merging a new sequence into the graph, this results in a new full path which is separated from the original vertices (Fig. 4.1)

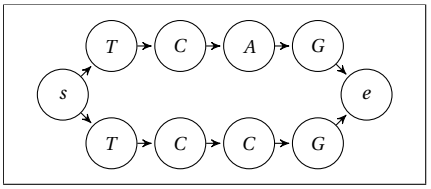
4.1.2 Building the index

There are two data structures needed for aligning a string against the graph: a suffix tree for left contexts and a suffix tree 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 (Appendinx 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. Two sets of strings, a left context set and a right context set, is generated for every vertice in the graph G. The generation of the two sets happen by the same procedure by swapping around the starting point and the direction of the iteration. When creating left contexts the algorithm starts in the start-node of G and traverses following the direction of the edges, for

	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 edit distance as a scoring schema



right contexts the opposite is done. Apart from this the two are equal. To generate the context set $c(n_x)$ for a given node n_x the algorithm looks at every string $c \in c(n_y)$ for every incoming neighbouring node n_y. Every c is modified into a new context string c' by trimming away the last character and prefixing the context with the character $b(n_y)$. All the generated strings c' is added to $c(n_x)$. 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 (see Fig. 4.2), 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 found 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.

The iteration starts in the node defined as the starting point which has the empty string ϵ as its only context. Whenever a node has finished producing its contexts it enqueues everyone of its outgoing neighbours in a regular FIFO queue. If a vertice has more actual incoming neighbours than incoming neighbours which are finished generating contexts, the node puts itself back in the queue. Thus happens to ensure that when a vertice is finished generating context, both the vertice itself and all preceding vertices are finished, and the remaining nodes can safely fetch contexts from its neighbours. The algorithm halts when the queue is empty. Every node has to be visited exactly once to generate its context, looking up its approximately b neighbours, and as the procedure runs twice to generate both sets the total runtime for the operation is $O(2|G|b)$.

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. **??**). 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 $O(|c|)$ (Discuss more efficient suffix possibilities somewhere?),

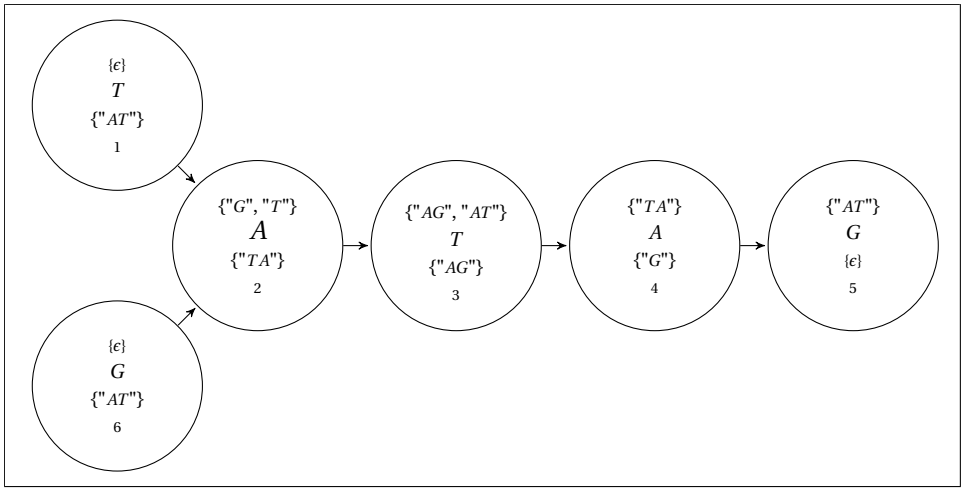


Figure 4.2: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

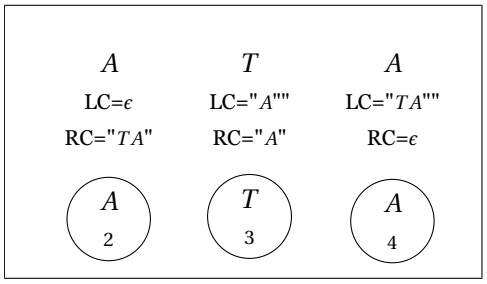
giving a total time complexity of $O(b^{|c|}|c|)$ per node per context set and $O(2|G|b^{|c|}|c|)$ for the entire graph. Building the entire index can thus be done in $O(2|G| + 2|G|b^{|c|}|c|)$.

4.1.3 Generating the modified graph

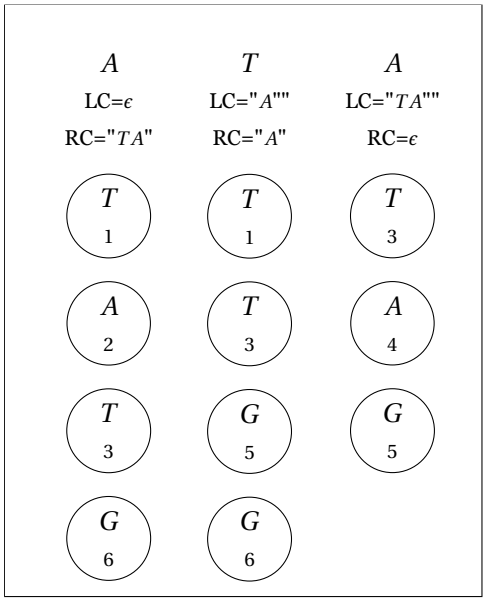
Creating G' is the process of determining which vertices qualifies as candidate vertices for a given input string s and how they should be connected. In order to determine actual candidates for the given string, the algorithm needs to know how much *fuzzyness* to allow. This is a measure which decides how different a read can be from its optimal counterpart in the graph before it is categorized as not mappable. The algorithm takes in a fuzzyness parameter λ which can be used to set a threshold $T = maxScore(x) - \lambda$. The maximal score is found by mapping the string x, be it the entire input string or a context string, against itself with the scoring function provided by the scoring schema. Both λ and T is used throughout the entire process as cutoff variables. Whether T is a threshold for the entire string, for a path or for a substring is either explicitly defined or unambiguous in the given context frame.

After generating the two context sets for every node, the elements of each one is inserted into their corresponding suffix tree. In theory every node can have $4^{|c|}$ contexts in each set. When the graph is more or less linear with few branches a more fair approximation is $B * |c|$ where B is the observed branching factor. The current implementation in the tool uses a naive suffix tree where insertion is $O(|c|)$. This is done for every node in the graph, yielding a total time complexity of $O(|G|B|c|^2)$. A discussion on more efficient suffix structures can be found in SOMEWHERE IN DISCUSSION. Every suffix is stored as a key with the index of it's originating node as a value. The total runtime for building a searchable index for a graph is $O(3|G||c|^2B)$
For every character $s_x \in s$ a left-context string and a right-context string is generated by looking at the $|c| + maxPossibleGapGivenFuzzyness(\lambda)$ surrounding characters. The two strings are treated as contexts, one left and one right, and used as a basis for a fuzzy search in it's corresponding suffix tree. The 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 contained in the child node (Reference actual code in supplementary?), (more explanation needed?). This newly created array is 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 threshold T for the given context string, every index contained in the node is stored as a pair on the form $\{index, score\}$ in the candidate set. If an index is stored several times, only the pair containing the highest score is saved.

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 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 4.4: The resulting candidate sets for mapping the string "ATA" against the reference genome from fig. 4.2 with varying T values

After the fuzzy search is concluded there are two sets of candidates for every index, one containing the nodes matching the left context and an equivalent for nodes matching the right context. These two sets 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. When the intersection happens the final set can again be pruned by removing all vertices which has a combined score that is lower than the combined threshold T for both contexts. When the vertices are found the edges need to be generated in order to finish the graph. Intuitively there should be an edge whereever there is a gap which is traversable without having the gap penalty exceeding λ. In practice this is a step which is done during the next step of the algorithm.

The newly formed graph G' can be defined formally:

$$G'(G, s, T) = \{V', E'\}$$

where V' is an ordered set of sets of length $|s|$ where each set V'_i is a set of nodes such that

$$V'_i = \{v_x | v_x \in G \wedge \exists [c \in c(v_x)](alignmentScore(c, c(s_i)) >= T)\}$$

and E' is a list of weighted edges such that

$$E' = \{e' = \{i_s, i_e, w\} | v_{i_s} \in V'_x \wedge v_{i_e} \in V'_y \wedge gapPenalty(y - x) <= \lambda \wedge w = distance(v_{i_s}, v_{i_e}) \wedge gapPenalty(w) <= \lambda\}$$

where $alignmentScore(x, y)$ and $gapPenalty(x, y)$ are scoring functions provided by the scoring schema and $distance(x, y)$ is the distance of the shortest path from node x to node y in the graph. (Mixing up nodes and indexes in the definitions)

4.1.4 Searching G' with a modified PO-MSA search

When the candidate nodes for each position has been chosen the next step is to find out how they can be combined into a single linear path. This is equivalent to finding the path through G' which traversal gives the best score within the given scoring schema. Conceptually this is in many ways similar to a regular PO-MSA search. The difference is that the roles are switched: Instead of searching through the reference graph with an input string we are searching through the indices of the string with the candidate nodes from the reference graph as our input. Instead of giving every node a score for every index in the string we give every index of the string a score for every candidate node. These scores are found through dynamic programming by filling out an array *scores* which has the same dimensions as the structure storing the candidate node sets. Because sets are not indexable, the indexes of the candidate nodes are also stored in an integer array $indexes$ such that $indexes[i][j]$ references the index for the j-th candidate node in the set V'_i and $scores[i][j]$ references the score for mapping the substring s' of s spanning the indexes 0 to i to the a path ending in the node with index $indexes[i][j]$. In order to store the actual path yielding the score a third array of pairs, $backpointers$, of the same dimensions, is also needed.

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 4.4 and $T = -1$

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

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

Then the nodes $n_x \in V'_i$ for the remaining candidate sets at the indexes $1 <= i <= |s|$ 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 variable looping over $indexes[i']$. Every entry-pair $((i, j), (i', j'))$ can be scored by a scoring function $\theta((i, j), (i', j'))$. The scoring function works by combining the score contained in the preceding entry, $scores[i'][j']$, a gap penalty, and a mapping score for the current index $mappingScore(n_{indexes[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length $distance(n_{indexes[i'][j']}, n_{indexes[i][j]})$. The final score stored in $scores[i][j]$

is the maximal achievable score θ produced by one of these pairs. $backpointers[i][j]$ is set to the to the index-pair (i', j') responsible for producing this score. The recursive formulas for the three arrays are defined by:

$$indexes[i][j] = x \quad \text{for } n_x \in V'_i$$
$$scores[i][j] = \max_{i',j'} \theta((i,j),(i',j')) \quad \text{for } 0<=i'<=i, 0<=j'<length(scores[i'])$$
$$backPointers[i][j] = \operatorname*{argmax}_{i',j'} \theta((i,j),(i',j')) \quad -||-$$

where θ is a scoring function defined as:

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

There are no restrictions in the recursive formulas to avoid alignments with aligned gaps in both the sequence and the graph. Because both gap penalties are counted the algorithm will however choose a path where only one gap is chosen in all scoring schemas where gaps are penalized with negative values, if such a path exists. By deciding the prioritized order of operations the algorithm can also handle scoring schemas with a gap penalty of 0. Within these types of schemas, the only scenario where an alignment with aligned gaps can be produced are scenarios where there are no valid candidate nodes with context scores larger than T for a subset of indexes i, which means there are no possible traversal of these nodes as a path which would give a score higher than T. This again means the path is not interesting because aligning it against the sequence would result in not mappable.

There are two components of the dynamic programming algorithm where a search is performed to find possible paths: The search backwards in the indexes and the search for distances between nodes in the graph. Both of these searches can be halted whenever the resulting gap penalty exceeds the parameter λ. This means the final time complexity is much more dependant on the tunable fuzzyness parameter than the size or complexity of the graph. The final time complexity for the entire dynamic programming step is $O(SOMETHINGIDIDNTPUSHTOGIT)$ (Appendix B).

Finding all optimal alignments

4.1.5 Handling invalid threshold values

An invalid threshold value is in this context seen as the cases where a string s is aligned against a graph G with a threshold T, but there exists no alignments for s and G which produces a score higher than T. By the definition of the problem this should result in s being unalignable which is equivalent to every element $s_x \in s$ mapping to no vertice in G. There are two cases in which this can happen: Either the algorithm finds an optimal path where the score is too low or there are no valid paths through G'. The fact that these two are the only cases can be proven by looking at their inverse: If none of the above are true there exists atleast one valid path through G and

the algorithm is able to find a path with a satisfactory score.

The first of the cases provides no problems programmatically as the search is able to find alignments, score them and determine which is the best. Why can this alignment not be output as optimal? Firstly one can argue this is done in order to provide consistency regarding the definition of the problem. Another, and more convincing, argument is that the search only find the best alignment present in G'. Already at the point of choosing candidate nodes actual elements of the real optimal alignment can have been pruned away as a result of not having a good enough context scores. Although these alignments do not fall within the strict definition they can be utilized for heuristical applications (Section 9.2).

Intuitively, the second case can be seen as instances where there are no paths from a set of startnodes indexes to a set of endnodes. Programmatically this is harder, as the edges are computed in real time. The algorithm recognices these instances when it finds a consecutive set of indexes which cannot be traversed without receiving a penalty larger than λ, and every vertice in every one of these candidate sets has no possible backpointers. Because of the possibilities of gaps at the beginning and the end of the alignment, the sets of start and endnodes can't be the candidate sets for the first and last index, but has to include all candidates for all indexes which are reachable without being penalized more than λ.

4.2 Merging aligned sequences

After aligning a sequence s against a graph G there exists an alignment A where every element $s_x \in s$ either has a mapping to an element $v_y \in G$ or does not map to anything. If we model s as a linear graph G_s with a single path, where the vertices have indexes corresponding to their position in s and edges corresponding to consecutive vertices, we can use the alignment as an equality relation such that $s_x \in G_s = v_y \in G \iff a_x = v_y$ to determine overlap between the vertices in G_s and G. The merged graph $G*$ will then contain a vertice set V_{G*} which is equal $V_G \cup V_{G_s}$, using the alignment as our equality determinator. Conceptually this means that every character c which maps to a vertice v is merged into that vertice, whereas every c which does not map to anything creates a new vertice. The edges of $G*$ is created the same way, however here there are 4 possibilites: Mapped vertice to mapped vertice, unmapped vertice to mapped vertice (and its inverse) and unmapped vertice to unmapped vertice. The 3 latter cases can all be generalized to an edge which is not already existing in G, because one of the vertices does not exist in G, and the operation is thus the same. The usage of the union-operator is critical because it ensures both that all new information from the sequence is stored aswell as the fact that the graph does not lose previously seen information, which would lead to deteriation over time.

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 common to both classes are described once in the section preceding the class-specific sections.

5.1 Common elements

5.1.1 Scoring schema

5.1.2 Hardware/runtime environment

5.2 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).

5.2.1 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.

5.2.2 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.

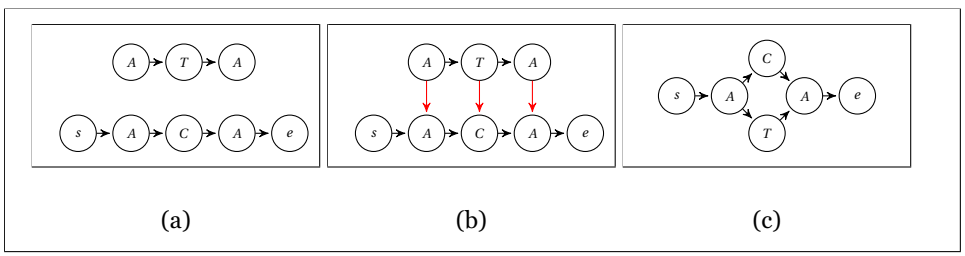


Figure 5.1: The syntax of the visual outputs. Two input sequences “ACA” and “ATA” are given, and the first is chosen as a basis for the reference graph (a). Then the second sequence is mapped against the graph (b) and merged (c)

5.3 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 2.2.3) 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].

5.3.1 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 2.1.5). why is this ok. In order to provide reproducability the randomness in the reads are generated from a seed.

5.3.2 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 15).

Chapter 6

Results

6.1 Proof of concept

6.1.1 Equal sequences

6.1.2 SNPs

6.1.3 Indels

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.