UiO : Department of Informatics
University of Oslo

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Esten Høyland Leonardsen

21st 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 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

This thesis is the final result of a master project in Informatics: Programming and networks. The project has been supervised by the Biomedical Informatics research group at the Department of Informatics, University of Oslo and was a part of the early phases of a larger project regarding graph-based reference genomes.

1.1 Motivation

We started out with an open exploration of the possibilities of using graph representations for reference genomes. Throughout the early phases the need for a formalization of the alignment process became apparent. Even something as basic as deciding what the graphs would look like could be solved through defining this process, by implicitly deciding how the input data should be compared and combined. The existing straight-forward approach was not satisfying, and alignment stood out as the most pressing problem to solve to get the ball rolling.

1.2 Aims of the thesis

The project in itself had a clear goal: Develop an algorithm for aligning against graph-based reference genomes. This thesis will not be concerned with the chronological events of the development process. Instead the thesis will be concerned with presenting the result of the project: The algorithm "Fuzzy context-based search". Interesting design choices taken throughout the process will be presented through the algorithm itself, the reasoning behind these choices given as formal arguments underway. Additionally, the thesis has two smaller goals:

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

In order to succeed with the two smaller goals, a tool was created which implements the algorithm. This tool is available online, instructions on re-

trieving and using the tool can be found in Appendix **??**.

Throughtout the development process we were faced with several decisions regarding the specificity of the problem. In many of these situations we chose to put an upper bound to the complexity, to end up with a simple, general, formally strict proof-of-concept, which should work as a base for later expansions into more specific applications. Some of these might seem as "shortcuts" to the reader: We assure this is not the case. Every time the result of one of these simplifications is presented we defend it. In the later parts of the thesis we reintroduce many of the when discussing the feasibility of the approach in relation to more specific biological problems.

During the master project the article "Canonical, Stable, General Mapping using Context Schemes" was published, discussing an approach to alignment which is similar to the one presented in this thesis. The similarities and differences between the two is granted a large part of the discussion section of the thesis.

1.3 Contents

The first chapter will present the theoretical background for the thesis: Why is there a problem which needs to be solved, what kind of data are we dealing with and more specific explorations of the previous progress on the subject. The third and fourth chapters are concerned with presenting the algorithm, first as a conceptual overview and then in more technical detail through explaining the implementation found in the tool. The next chapter addresses the validation of the algorithm. This is done through a series of examples run on the tool and can thus also be used as an introductionary manual. Chapter 6 tests the performance of the tool by running a large number of tests on large, real-life datasets. The results from both these chapters are discussed in Chapter 7, along with the previously mentioned comparison of two approaches to the problem. In the eight chapter a conclusion is drawn regarding the value of the approach before finally possible future improvements is discussed in Chapter 9.

Chapter 2

Background

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

2.1 Genetics

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

2.1.1 The central dogma

The process of transforming the genetic information into large functional biomolecules is called *the central dogma* of molecular biology. The central dogma states that DNA is transcribed into *messenger RNA* (mRNA) which in turn is translated into proteins. mRNA is, like DNA, a sequence of nucleotides consisting of the three bases A, C and G and *Uracil* (U) instead of T. The mRNA can be divided into triplets of nucleotides called *codons*. The cell decodes the mRNA codons and create strings of amino acids which are transformed into functional proteins. The relationship between codons and amino acids can be lookep up in a table called *The standard genetic code*[16, Chapter 1, p. 6]. Only a portion of the nucleotides in DNA act as *coding regions* which make it through the transcription process and code for actual protein sequences. These are also called *exons*. The remaining *non-coding regions* of the genetic sequence are known as *introns*. In humans about 1.3% of the genome is coding regions[16, Chapter 4], the rest used to be referred to as *junk DNA*. We now know that the non-coding regions also holds important information.[REF]

2.1.2 Variation

Genetic information is prone to mutations, either as a result of environmental influence or as a consequence of imperfections during DNA transcription. The simplest mutations are *point mutations* which affect a single nucleotide base. Point mutations can either be *Single-nucleotide polymorphisms* (SNPs) where a single base is substituted for another, or *insertions* or *deletions* (indels) where a single nucleotide is removed or inserted into the genetic sequence. Mutations can also occur over larger areas of the genome, where longer subsequences can be deleted, inserted, moved or reversed. A final type of mutations is *Copy number variations*, or *repeats*, where a longer sequence of DNA, typically at least 1 kb [7], is repeated a variable number of times.

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

Because there are more possible combinations of nucleotide triplets than there are amino acids there exists some overlap between the codons and the resulting amino acid. For instance the DNA triplets “CGA”, “CGC”, “CGG”,“CGT”, “AGA” and “AGG” all encode for the amino acid Arginine.

In these cases point mutations can occur without affecting the resulting protein. These mutations are called *synonymous*, the opposing case which alters the amino acid sequence are called *non-synonymous*.

2.2 Genetic data, sequencing and string algorithms

As genetic information is vital for determining the function of an individual, there is an obvious wish to understand this data. A large set of technologies and methods have been developed to collect and analyze the information. This section will present some of the most important concepts in handling this data and the field of bioinformatics. We will also present the MHC region, a genetic region which we used when testing our approach.

2.2.1 Reference genomes

A *reference genome* is a data structure which contains genetic information for a population, typically for a given species. The reference genome has a set of continuous nucleotide sequences, called *contigs*, combined into larger *scaffolds* which again are combined to form the *genome* for a species. The first reference genomes collapsed samples from several individuals into a linear *consensus sequence* which was representable for the species as a whole. Later reference genomes have been built more flexibly to allow positions on the genome, called *loci*, to have several variants, termed *alternate loci*. A specific variant of a gene is called an *allele*. A *haplotype* is a set of alleles which tend to be inherited together. Reference genomes form what can be seen as an index for the genome of a species and can be used in sequencing[1]. The structure of a reference genome can increase computational tractability compared to storing a set of indivvidual genomes, by reducing the double-storage of equal regions. The reference also provides a mosaic representing genetic variation, which can be useful when doing genetic analysis.

2.2.2 The human genome

The human genome consists of roughly 3gb. The base pairs are spread over 46 chromosomes and are assumed to contain about 23 000 genes [16]. The human reference genome is developed and maintained by the *Genome Reference Consortium*[9], the current version is called the GRCh38[10]. GRCh38 contains 261 alternate loci, spread over 178 out of a total of 238 regions. An average human is estimated to deviate from the reference genome in 10.000-11.000 synonymous sites and 10.000-12.000 non-synonymous sites [5].

[1]Covered in section 2.2.3

Major Histocompatibility Complex

The *Major Histocompatibility Complex* (MHC) is a genetic region spanning approximately 4.5-5 million base pairs (mb)[6][25]. 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 [33]. Even more so MHC is known to be a highly variable region, containing variants that are directly associated with disease [11]. The high variability creates difficulties when comparing DNA sequences to determine genetic causes for the observed disorders.

2.2.3 Sequencing

During *sequencing* a *sequencing machine* is used on a physical DNA fragment to find the underlying nucleotide sequence. The machines produce short *reads*, typically in the order of a hundred bp[29] 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[28] or de Bruijn graphs[2] are often used in what is known as *de novo assembly*[16, Chapter 1, p. 19].

The different sequencing technologies have varying degrees of errors introduced in their reads, often closely related to the sequencing cost[29]. 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 [39] and correcting the reads[20] Can probably provide more citations.

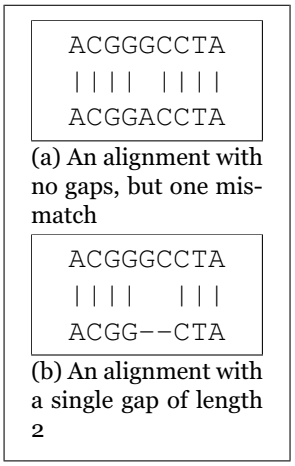
2.2.4 Alignment

Sequence alignment is the process of determining correspondence between text strings, in this case representing DNA, by mapping the elements from one to the elements of the other according to a *substitution matrix* (Table 2.1) providing a *mapping score*. Throughout this thesis we will let the notation $mappingScore(c_1, c_2)$ denote the score for mapping two characters c_1, c_2 against eachother. Alignment of DNA strings has several important applications, as previously mentioned it is utilized as a tool for assembly as well as in genetic analysis and comparison.

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

[2] Presented in 2.3.1

let the notation $gapPenalty(distance)$ denote the gap penalty achieved for a gap of length $distance$. Gap penalties come in different shapes, often according to the origin of the data involved. A *linear gap penalty* gives linear penalties related to the gap length. An *affine gap penalty* distinguishes between opening and continuing a gap. A *logarithmic gap penalty* lets the increase in penalty fade as the gap expands. We let which one of these is used, along with the choice of substitution matrix be defined in a *scoring schema*. A scoring schema can thus also be seen as any structure which provides the $mappingScore$ and $gapPenalty$ functions. The scoring schemas can be based around simple match/mismatch scores, which corresponds to the mathematical *Edit distance problem*[3], or more complex scores (Table 2.1). These complex models typically try to model the probabilities behind the physical processes responsible for change. The computational sequence alignment problem consists of finding the highest scoring alignment for any two strings. There exists two main variants of the problem: Finding *global alignments*, where two entire strings are aligned against each other, and finding *local alignments*, where a string is aligned against a substring of the other. The two are traditionally solved respectively by the Needleman-Wunsch and Smith-Waterman algorithms which both are based on *dynamic programming*[4].



(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 are expected to share a common ancestor. The goal is to determine which traits in the individuals arised from the same origins and how the involved species have diverged genetically over time. A final variant of the alignment problem is one involving large databases of sequences, where the algorithms does not only need to find the best alignment between two sequences, but also determine which sequence should be chosen in order to maximize the result. Both of the preceding techniques typically utilize heuristical methods in order to decrease the computational complexity.

[3]Presented in detail in the next section
[4]Also presented in the next section

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

Table 2.1: The HOXD70 substitution matrix

2.2.5 Dynamic programming

Dynamic programming (DP) is a problem-solving technique where a problem instance is solved by breaking it into smaller subproblems and combining their results. DP is similar to recursion in that every instance is solved by a *recurrence relation* (An example can be seen in equation 2.1) which recurses on smaller and smaller problems until a *base case* is found. A base case represent the bottom of the recursion and is a value which can easily be computed without further lookups. The main difference between recursion and DP is that the latter usually stores its intermediate results to allow for fast lookups for 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 (ED). ED is concerned with finding the minimal amount of substitutions, deletions and insertions needed to transform one string into another. The algorithm utilizes a 2-dimensional array to store the computed values (Figure 2.2). For two strings S and P, every index $[i, j]$ in the edit distance table represents the problem instance of the substrings $S[0:i], P[0:j]$.

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

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

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

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

There are two distinct ways of utilizing Dynamic Programming. A *bottom-up* approach starts at the smallest cases and computes everything until it reaches the actual given problem instance. This corresponds to starting in the top left corner of the edit distance array and computing the cells iteratively moving downwards to the right. A *top-down* procedure starts at the given problem instance and recursively computes every subproblem that is needed. This means starting in the bottom right corner of the 2-dimensional array and recursing upwards to the right. For the edit distance problem the choice of approach bears no big significance as every cell has to be computed either way, but there are problems where using top-down can avoid some computations which are irrelevant to the final result. The latter can also be efficient for heuristical methods where an area of the search space can be overlooked.

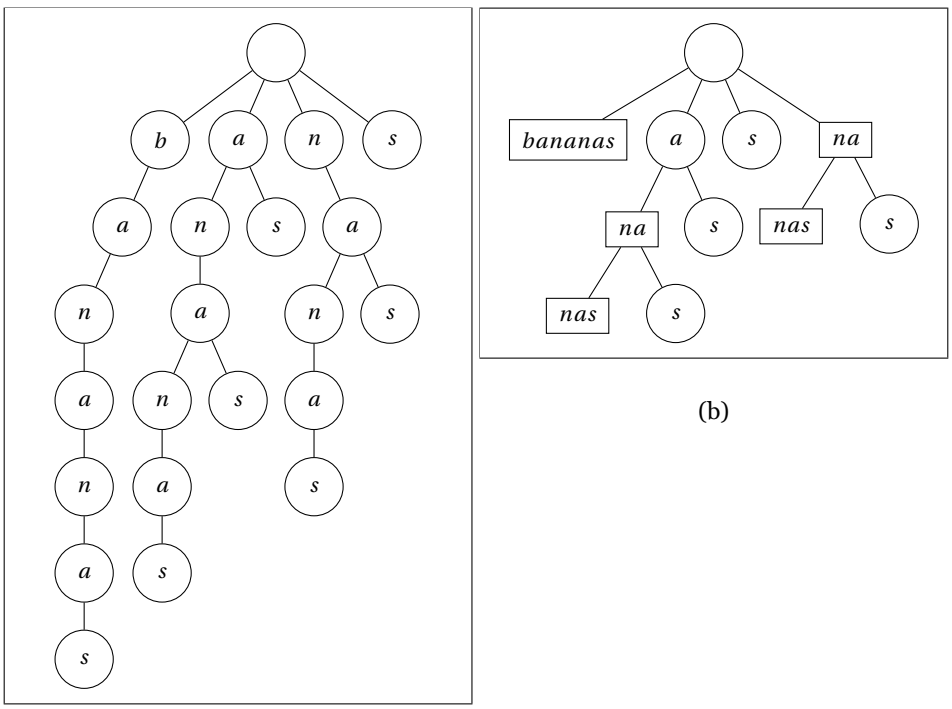
2.2.6 Suffix trees

A *suffix trie* is a special tree constructed specifically for strings of text, containing vertices representing characters (Figure 2.2a). When creating a suffix trie for a given string, every suffix has a corresponding leaf vertice such that where the vertices along path from the root to the vertice contains the characters of that suffix. Consequently, every substring has a path starting in the root node. A *compressed suffix trie*, or*suffix tree*, is a suffix trie in which every linear path is compressed into a single vertice (Figure 2.2b). Both suffix tries and suffix trees can easily be extended to hold collections of strings[1, Chapter 20]. A simple implementation has a space complexity of $O(s)$, where s is the length of the string (or the total length of all strings if the tree is built from a collection), and a string of length m can be looked up in $O(km)$ time for an alphabet of size k[1, Section 20.6.1]. The tree can be constructed in linear time[36].

2.2.7 Compression

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

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



(a)

(b)

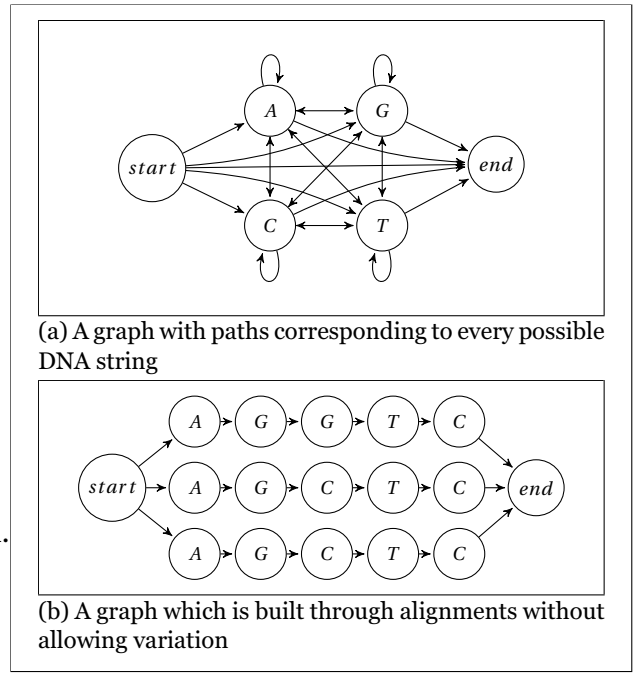
Figure 2.2: The suffix tree (a) and suffix trie (b) of the string “bananas”

2.3 Graph-based genome representations

In the previous sections we have been introduced to the variable nature of genetic data. The linear model provided by text strings seems suboptimal for representing this variation due to its innate lack of flexibility. In this section we will present graphs as an alternative model for genetic data. Graphs are far more expressive, and thus able to represent more complex relationships between the elements involved. If we are able to rephrase biological questions in graph theoretical settings, we can also benefit from the extensive mathematical field of graph theory when searching for solutions to the arising problems. However, when changing the underlying structure a major problem appears: In order to avoid a drop in functionality, the more complex model calls for more sophisticated variants of existing methods. Graph-based approaches have been used for some time in the assembly process, and more recently in relation to reference genomes. We will present the work done on both these subjects alongside what we see as some of the remaining unsolved problems. The contents of the section is presented in a way which should not require any previous knowledge of graph theory beyond elementary terms, but readers interested in a more complete introduction is referred to the bibliography[40, Chapter 9][1, Chapter 11][32, Chapter 0]. Complexity in regards to the graphs and their operations is discussed using *big-O* notation[40, Chapter 2][1, Section 3.1]

2.3.1 Model

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



(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 propsed graph models displaying flexibility (a) and rigidity (b)

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

De Bruijn graphs

In the article "An Eulerian path approach to DNA fragment assembly"[28], Pevzner, Tang and Waterman proposes *de Bruijn* graphs as a solution to the problem of finding the correct assembly of repeats during fragment assembly. A de Bruijn graph is a structure where vertices represent *k-mers* from an alphabet and edges represent relationships between the k-mers of two vertices (Figure 2.4a). Pevzner et al. lets the vertices contain strings of length $l-1$ and connects vertices with an edge wherever there exists a read of length l containing the two substrings. Formulating the problem in this fashion makes us able to see it as a *Eulerian path* problem, solvable in polynomial time, rather than the traditional "overlap-layout-consensus" method which is equivalent to the NP-complete problem of finding a *Hamiltonian path*[1, Section 11.1]. A great benefit with de Bruijn graphs is that there is no disambiguity: Any legal k-mer has at no point more than one vertice representing it.

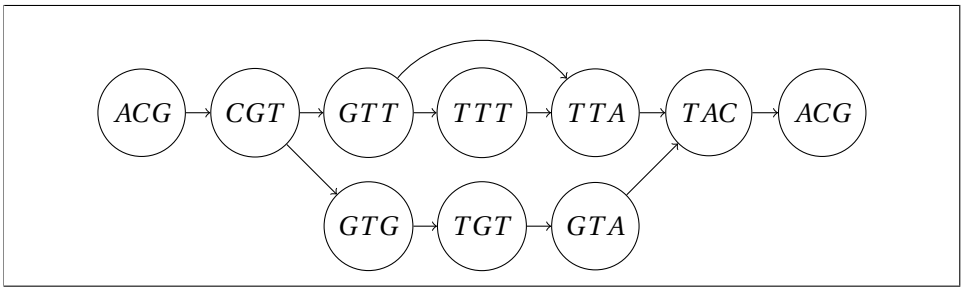
A more detailed type of de Bruijn graphs is the colored variant where the origins of edges and vertices are stored as colors. The entire sequence originating from a single individual sample can be seen by following a path with a given color. Similarities between samples can be seen as multicolored stretches,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

Sequence graphs

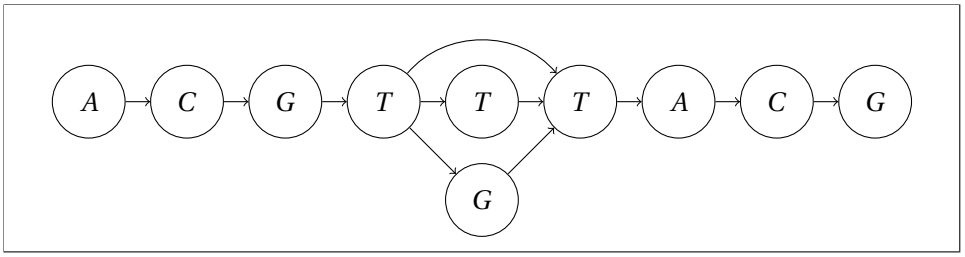
The relationship between a de Bruijn graph and the sequences it represents is not immediately apparent. A more intuitively pleasing representation is a graph where every vertice contains exactly one nucleotide (Figure 2.4b), a concept called *partially ordered graphs* by Lee et al.[6] and *sequence graphs* by Paten et al.[26]. 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 as propsed by Paten et al., for uniqueness. Even though these ids can be used to identify a vertex they contain no information regarding the relationships between the elements. The difficulties presented by this problem will be the basis for the subsequent section on mapping.

Cactus graphs

The article “Cactus Graphs for Genome Comparisons”[27] 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 (Figure 2.4c). In cases where there exists several adjacencies between two vertices these are combined into a single edge with several labels. The result is a graph where every *simple cycle*, cycles where no 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 and thus the strictness of “equality” between strings represented in a single vertex. If the strictest possible restriction is set, a requirement of equal strings, the vertices would contain an exact k-mer from an input sequence just like the de Bruijn vertices.



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



(b) Sequence graph

(c) Cactus graph

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

2.3.2 Mapping

One of the previously mentioned operations which can control the properties of the model is mapping. Although the two terms are often used isomorphically we will in this thesis define mapping and alignment as two separate concepts. We let mapping be the process of finding relationships between single characters of a string and single elements of a reference genome. Alignment is concerned with finding relationships between consecutive elements of an input string and substructures in the reference genome. In comparison of linear strings, mapping is easy: Every string has the same underlying coordinate system, represented by the positions of the characters. This means two elements from two separate sequences are either in the same position or they are not. When they are not, the difference in position can be derived from the difference between the indexes. If we assume the indexation system proposed in the last section, the indexes of a graph has only one property: Uniqueness. They do not hold the intrinsic value of describing relations between vertices. Any mapping system which uses fixed coordinates would face problems when dealing with a fluent graph able to merge in new information, as the internal relationships are bound to change. In de Bruijn graphs the problem is solved by moving the mappable quality away from positions and into the data: For any possible k-mer there either is a corresponding vertice or there is not. In sequence graphs, where nucleotides are the most basic information, there exists an equal number of identically scoring positions for every base as there are vertices containing that vertice in the graph.

Paten et al.[26] introduce the concept of *context-based mapping* as a solution to the mapping problem when the reference is modeled as a sequence graph. Context-based mapping is an approach where a vertex is identified by the surrounding environment in the graph. More technically a vertex has a set of *contexts* which are tuples (L, B, R). The L references the left side, a path going in to the vertice, B is the base contained in the vertice and R is an outgoing path from the vertice (Fig. figure). More conceptually, contexts can be seen as paths which pass through a given vertice. Because these paths are linear and passes through vertices containing characters, the contexts can be treated as text strings. There are two concrete examples of approaches presented in the article: The *general left-right exact match mapping scheme* and the *central exact match mapping scheme*. The key words left-right and central refer to how a vertice defines its contexts based on the 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, meaning

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 a *context set*. Because every context is unique a collection of such will also be unique, which means context-based mapping leads to a two-way unique mapping schema. This is an even stronger notion of mappability than positions in strings, as a character of a string does not necessarily map uniquely back to its position. Being this precise in the definition has a drawback: If a vertice does not have a unique context it is no longer mappable. In spite of this the context-based approach presents a precise and efficient solution to the mapping problem. This is knowledge we will bring with us when we move on to considering the more complex alignment problem.

2.3.3 Alignment

As previously discussed, alignment of text strings has for some time been considered solved. We let the two strings represent each their dimension in a two-dimensional space and search for a path through the space which yields an optimal score. When one of the strings is replaced with a graph a simple two-dimensional representation is no longer sufficient. The "3 steps before" or "11 steps after" relationship found in strings is no longer as easily derivable. A solution to this problem can be to imagine alignments 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"[14] proposes a direct adaptation of the regular two-dimensional dynamic programming solution for graphs through the *Partial Order Multiple Sequence alignment* (PO-MSA) algorithm. Every vertex contains a one-dimensional array representing the string which is to be aligned. Just like an array-index in the edit distance problems, the 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, it is no longer sufficient to look up the preceding index. The vertex has to look at every preceding vertex as a single instance of the two-dimensional problem, to determine which of the incoming vertices represents the linear path which presents the highest score. After filling out every index i of the array in the vertex v in this fashion, the array represents the highest score possible for the substring $S[0:i]$ for all paths ending in v.

Using an approach which is this closely related to the known approaches for regular string alignment has its advantages. Alignments and scores are verifiable through existing tools and the principle of optimality is contained

through the dynamic programming. Techniques for handling the different types of alignments, for instance local or global, can be inherited from the domain of strings. The algorithm is however a crude adaptation and thus susceptible to the inherent complexity of graphs.

Context-based alignment

In the article "Canonical, Stable, General Mapping using Context Schemes"[25] the previously mentioned concept of context-based mapping is used for aligning entire strings. The algorithm works by identifying substrings of the input string which 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*. The introduction of the α and β variables allows for a regulation of the strictness of uniqueness and presents a powerful approach for alignment against complex reference structures. However it is still a heuristic, based on a non-tautological assumption of the involved input data.

The remainder of this thesis is concerned with presenting our attempt of pursuing a more formally strict alignment algorithm, developed partly at the same time as the approach presented in this subsection. We will return to discussing the similarities and differences between the two in Chapter **??**.

Chapter 3

The algorithm “Fuzzy context-based search”

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

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 through definitions of their constituents.

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

The involved graphs will be sequence graphs[1] where every vertex correspond to a single nucleotide from a one or more input sequences used in building the graph. Whether the vertex originates from a single or several sequences is based on whether any new bases has been mapped, and consequently merged, into the vertex. In addition to the nucleotide the vertices will contain an index which is unique to the graph.

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

Every graph G will have two special vertices $s_G = \{s, 0\}$ and $t_G = \{e, -1\}$ which represents unique start and end vertices.

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

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

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

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

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

Paths spanning the entire length of a graph G, from s_G to t_G, are named full paths. Every input sequence used to build the graph has a corresponding full path.

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

[1] Discussed in section 2.3.1

There is no correspondence the other way, meaning there can exist full paths which does not originate from a single input sequence. An example of this can be seen in figure 3.1 where a reference graph made from three sequences has 9 valid full paths. When aligning regular text strings against each other the introduction of gaps is a key element. We translate this concept to the graph domain through the introduction of *incomplete paths*.

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

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

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

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

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

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

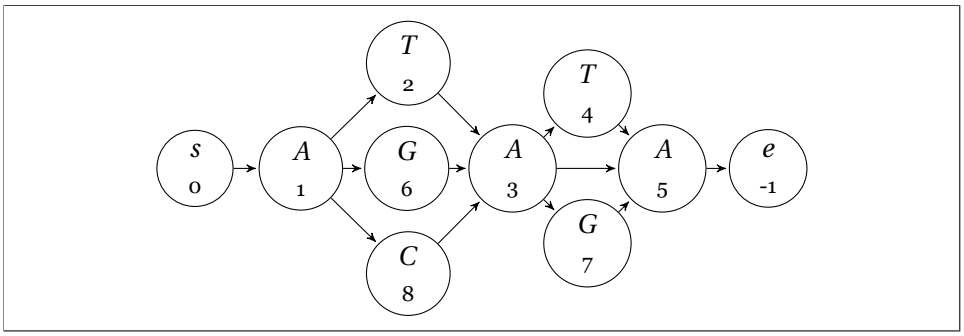


Figure 3.1: An example reference graph G made from the three sequences "ATATA", "AGAGA" and "ACAA"

3.2 The alignment problem

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

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

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

Once we have a clear definition of a graph G and an input sequence s we can specify what an *alignment* between the two should look like, representing a model of the relationship between them. In order to achieve this goal, the alignments should provide relations between the smallest constituents of the two input structures, the vertices of the graph and the characters of the string, in a way such that the 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 9 (Alignment)
Given a graph G and a string s, an alignment A is an ordered list of length $|s|$ such that every element $a_x \in A$ is either 0 or the index for a valid vertice of G such that for every consecutive pair of valid indexes a_n, a_m there exists a path P where $p_0 = a_n$ and $p_{-1} = a_m$. A 0 represents an unmapped character in s.

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

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

The first two can be looked up through the standard functionality provided by the scoring schema and equation 3.1. The last can be found by summing

up the gap penalties for all the gaps in the input sequence. A gap in the input sequence can be identified by a continuous subsequence $A*_{x:y} \in A$ spanning the indexes x to $y-1$ where every element is unmapped. An important aspect here is that every unmapped element should only be considered part of exactly one gap. We cover this aspect by only considering *maximal unmapped subsequences*

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

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

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

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

Definition 11 (Alignment score)
Given a sequence s, a graph G and an alignment A, the score produced by combining mapping scores for the pairs $\{b(v_{a_x}), s_x\}$ for $0 <= x < |s|$ 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 our graph-based adaptation[2] of the alignment problem:

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

Notice that the definition only calls for finding one of the alignments which produce the highest possible score. This is done in order to simplify the conceptual explanations of the algorithm. Implementation-wise this can trivially be changed to finding all optimal alignments. The necessary adjustments is discussed as a part of the succeding chapter in section 4.1.3.

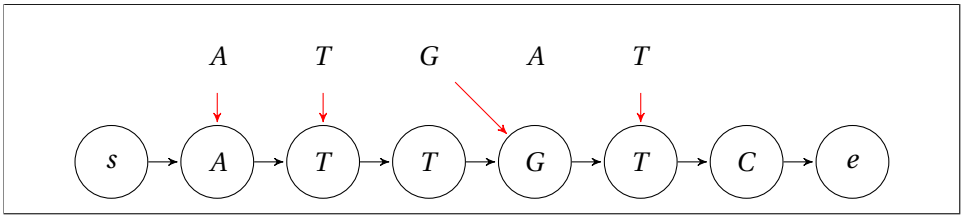


Figure 3.2: A visualization of an example alignment of the string "ATGAA" against a reference genome made from the string "ATTGTC". The actual alignment is visualized through red arrows pointing from the characters of the string to the vertex they are mapped to. We see a gap in the incomplete path between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

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

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

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

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

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

3.3 "Fuzzy context-based search"

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

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

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

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

3.3.1 Constructing the candidate graph

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

We first let V' be a subset of the original vertex set V. In order to guarantee optimality we put a restriction on V':

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

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

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

1. Every $v_x \in V$ should be in V'_y if there exists an optimal alignment A with a higher score than T where $a_y = x$

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

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

If we manage to create V' from these two restrictions we can guarantee a 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 being related solely to the computational complexity, which means it does not have to be strictly enforced. We can thus relax it without interfering with the principle of optimality:

2. Every vertice $v_x \in V$ 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 complex or the result is too inaccurate. We can let the second restriction be an informal description of a search for an optimal middle ground between the two:

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

The rest of this section will describe our method for approximating this middle ground

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

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

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

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

Definition 14 (Graph genome weighted edge (Weighted edge))
A triplet $e' = \{i_s, i_e, w\}$ where the two first elements are indexes for vertices in V' and the latter is an integer

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

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

The resulting graph is still very complex . Every 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 creates an upper bound both on how far back in the candidate sets a vertice looks for neighbours and, more importantly, 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 exhaustive dynamic programming. The search algorithm is conceptually very similar to PO-MSA[4], except the roles are switched around: Instead of searching through the reference graph with an input string we are searching through the indices of the string with the canditate vertices from the reference graph as our input. When we dynamically compute scores we are still doing the same thing as a regular PO-MSA, letting a candidate vertice v_x in a candidate set V'_i be an intersection at position x, i in a two-dimensional space where the dimensions represent the string and the path. We let an individual score identified by x, i be the highest possible score for aligning the substring $s_{0:i}$against a path ending in v_x. In this way we can find the highest possible score for the entire alignment in the highest scoring node in the last candidate set.

The base case of the dynamic programming are the candidate vertices in the first candidate set, $v_x \in V'_0$. We initialize these scores to $mappingScore(b(v_x), s_0)$, which is equivalent to aligning them against the substring containing exactly the first character of the string. During the following bottom-up procedure we will be faced with another set of base cases: Vertices which have no incoming edges. If the vertices are reachable by gapping over the preceding indexes of the string without the gap penalty exceeding λ we initialize them to their mapping score combined with the gap penalty. In all other cases we set the score to an arbitrary low value, for instance $-2 * \lambda$, which yields any alignment starting with the vertex a score lower than T. This means they can not be considered candidates for the optimal alignment.

The recurrence relation of the dynamic programming algorithm is concerned with setting the score for any 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, all edges traversed in the final alignment will only be penalized

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

for one of them. We know this because whenever there exists an alternative with only one gap, this will be prioritized due to a lower gap penalty. Whenever there does not exist such an alternative, this means the candidate vertex which "should" have existed is not a member of any contexts scoring high enough, meaning the path is never interesting when creating an alignment with a score higher than T.

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

3.4 Heuristical applications

In this chapter we have presented a strict non-heuristical algorithm which is guaranteed to give the correct results whenever it is fed the correct parameters. This is a concept which provides formally steady ground when discussing the tractability of the conceptual solution to the problem. When nothing else is specifically stated, the remainder of the thesis will be concerned with the approach presented up to this point, in all its formally provable glory. However, we now want to take a brief detour to examine some of the heuristical possibilities which has emerged. In the approach, if we are not able to find a good enough score for a vertex in the last candidate set we simply classify the string as unalignable. This might seem a bit rash, as we have already completed a strenuous search for similarities between the two input entities. As previously mentioned this is a design choice taken to land at an easily perceivable, formally strict, proof-of-concept algorithm. If one were to imagine the strict optimality restriction were dropped, it is fair to assume that there exists a set of "good" solutions in the combination of the entities we have found so far. A more specific explanation on the details of doing this transition can be found in section 4.2.

It might seem silly deeming strings unalignable when the exhaustive search done in the second part of the algorithm is bound to find an optimal alignment through the vertices found in the first step. The reason for this hard cut-off is the similarily hard pruning done during this first search for candidate vertices. The pruning is done strictly as a function of λ: If we start utilizing a different error-margin during the final search we can no longer guarantee this search is done on a complete set of candidate vertices. This means an optimal alignment for the candidate vertices is no longer equivalent to an optimal alignment for all vertices.

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

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

4.1 Aligning sequences

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

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

4.1.1 Building the index

There are two data structures needed for aligning a string against the graph: A suffix trie for left contexts and a suffix trie for right contexts. Before either of the two are built the algorithm needs to decide a length for the contexts. 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 **??**). The length of the contexts does not impact the quality of the alignments found by the algorithm (Shown in the proof in Appendix **??**) but will have an impact on the runtime (Calculations can be found in Appendix **??**).

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

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

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

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

and t_G swaps places as start and end-vertices, the definitions of what is an incoming and an outgoing neighbour is switched 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 (Figure 4.1), and thus avoid explosive exponentiality in the context set sizes. Unlike the method of Paten et al. [26] there are no requirements for contexts to be uniquely mappable to exactly one vertice. Because the last step of the algorithm does a search for an incomplete path through all the candidate vertices this presents no difficulties when finding the alignment. Furthermore, dropping this precondition assures every node has two valid contexts and are thus present in both suffix trees.

After generating the two context sets for every node, the elements of each one is inserted into their corresponding suffix tree. Every suffix is stored as a key with the index of it's originating node as a value (fig. 4.2). In theory every node can have $4^{|c|}$ contexts in each set, in practice a more fair approximation is $b^{|c|}$ where b is the observed branching factor for the graph. If we assume our graphs are too a large degree linear, we can assume $b = 1$ and thus $b^{|c|} = 1$[2]. The current implementation uses a naive suffix tree implementation where insertion is $O(|c|)$, giving a total time complexity of $O(b^{|c|}|c|)$ per node per context set and $O(2|G|b^{|c|}|c|)$ for the entire graph. The generation process visits $|G|$ vertices once in each direction, which means the entire index can be built in $O(|G||c|)$[3].

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

[3] Assuming $b^{|c|} = 1$. Further explanations of simplifications done in big-O notation can be found in the bibilography[40, Chapter 2][1, Section 3.1]

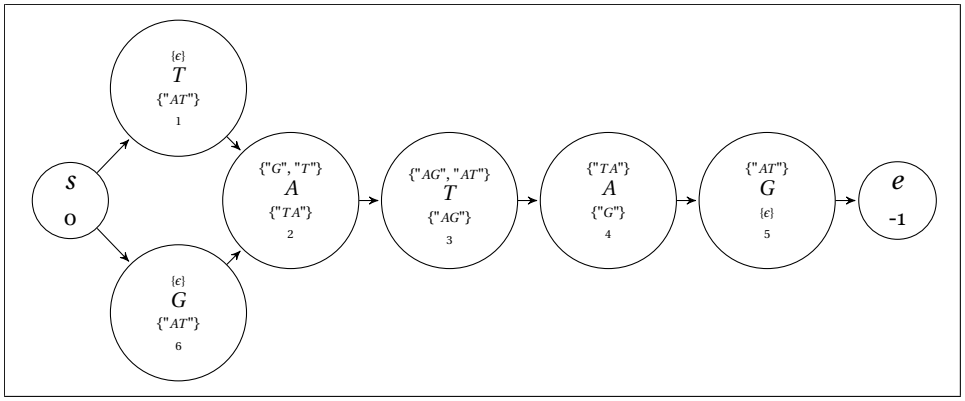


Figure 4.1: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

4.1.2 Generating the modified graph

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

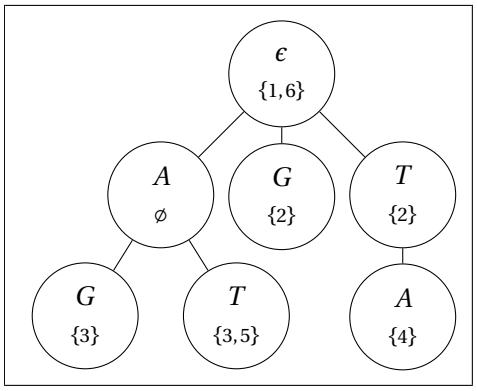


Figure 4.2: The left suffix tree corresponding to the graph in 4.1

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

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

scoring array corresponds to mapping the entire string c against the entire context achieved by concatenating the characters contained in the path through the tree traversed by the recursion. If the score is higher than the context threshold T_c for the given context string, every index contained in the vertex is stored as a pair on the form $\{index, score\}$. The candidate sets are implemented as maps with the index as a key, which allows us to store the index of every candidate exactly vertex once by only saving the pair which produces the highest value.

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

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

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

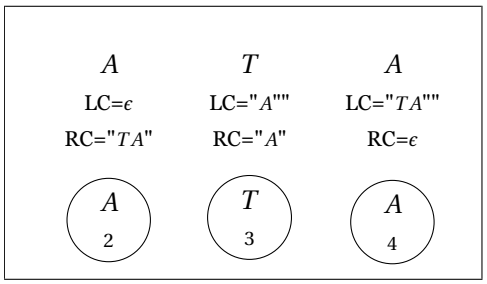
After the fuzzy search is concluded there are two maps of candidates for every index, one containing the vertices matching the left context and an equivalent for vertices matching the right context. The keysets of these maps are intersected to produce a final candidate set for the index i, where the score is created by adding together the scores from the two original sets. During the intersection process the final set can again be pruned by removing all vertices which has a combined score that is lower than the combined threshold T_c for both contexts. If one of the sets are empty, due to pruning in the suffix tree search, we simply emulate the intersection by keeping the non-empty set. Formally we can define the new vertice set V'

[4]See calculations in Appendix **??**

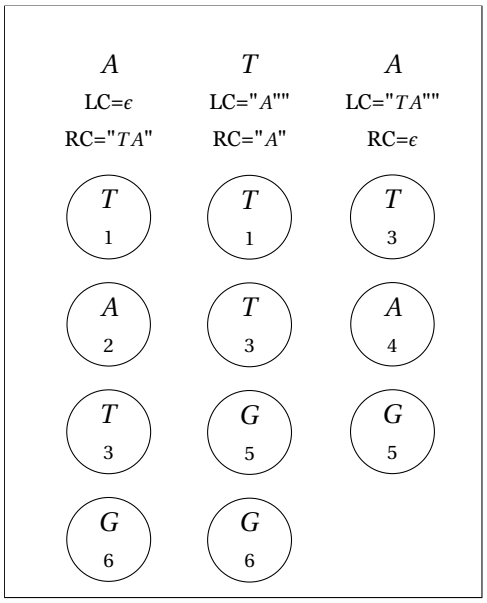
as an ordered list of sets V'_i for $0 \leq i < |s|$ where

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

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



(a) $T = 0$



(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

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

4.1.3 Searching G' with a modified PO-MSA search

Once we have created the graph we can start the setup for the subsequent search. We move the indexes of the vertices in the candidate sets over to a two-dimensional array $indexes$ where we let one dimension represent the indexes of the string and the second the individual vertices in the candidate set. This is done in order to have the vertices in a structure where we can reference them solely by an index. In addition to the $indexes$ array we create a floating point array $scores$ with exactly the same dimensions, to contain the scores for the individual vertices. Additionally, in order to backtrack and find the optimal alignment, we create an equally sized $backpointer$ array. Conceptually this should store pairs of integers referencing the other indexes in the array, in the tool this is implemented using strings.

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 internal relation which we want to preserve. However, from this point onward, we have given the elements an order. Although the internal ordering does not hold any value this is important as we now know the same index in the three separate arrays reference the same vertex.

After setting the values in the easily identifiable base cases we start computing scores for the paths in our graph. The nodes $v_x \in V'_i$ for the remaining candidate sets at $1 \leq i < |s|$ are looped over with j as a counter, and $indexes[i][j]$ is set to x. For every such entry a list of pairs is made with other indexes (i', j') such that i' is a preceding index $i' < i$ and j' is another counter variable looping over $indexes[i']$. For every entry-pair $((i, j), (i', j'))$ we produce a score by the scoring function $\theta((i, j), (i', j'))$. The scoring function works by combining the score contained in the preceding entry, $scores[i'][j']$, the gap penalties, and a mapping score for the current index $mappingScore(v_{indexes[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length $distance(v_{indexes[i'][j']}, v_{indexes[i][j]})$. The computation of the last gap penalty corresponds to finding the edges, which up to this point have been without interest to the algorithm. We search for these distances by a breadth-first regular graph search which starts in the vertice $n_{indexes[i'][j']}$ and is concluded when we find $n_{indexes[i][j]}$. Whenever we search for more than λ steps without finding the target vertice, we can return the current distance multiplied by 2. When a gap penalty is computed for a gap this length the score will always be $2 * \lambda$ which means the edge can never be part of a final alignment and is thus not interesting.

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

three arrays are thus:

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

where θ is a scoring function defined as:

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

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

Finding all optimal alignments

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

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

branch represents several solutions with the same score, and the read is thus classified as unalignable.

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 unmapped.

Yielding an empty alignment for the first case might seem strange as the algorithm does a full computation and finds a legal path. There is however one important consideration: Vertices which are parts of an optimal path might have been dropped during the pruning of the candidate sets. There is one very interesting case where this occurs, namely when the difference between the threshold T and the highest scoring alignment is exactly 1 (or the minimal penalty possible for generalized context 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 regions which are aligned well within the threshold.

4.2 Necessary adjustments for a heuristical search

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

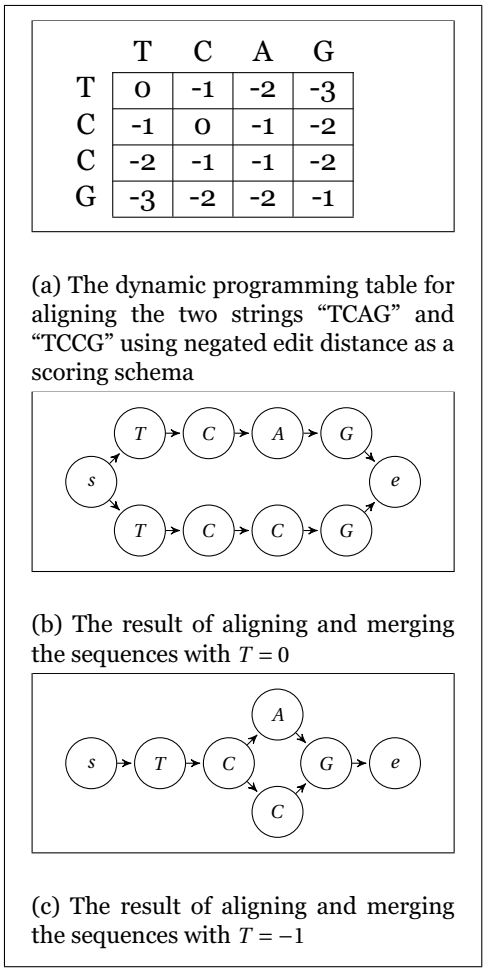
over actual paths. This can be done by increasing the arbitrary value to another large number, for instance $gapPenalty(|s|)$. If we want the heuristic algorithm to find incomplete paths which does not previously exist in the graph, by jumping between branches or horizontal positions, we would need to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

4.3 Parallelization

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

4.4 Merging aligned sequences

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



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

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

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

Figure 4.4: Different scoring thresholds T yields different reference graphs

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

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

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

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

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

There is one important aspect to be considered when using the merge procedure: Every sequence which is merged in is considered a standalone sequence which starts at the beginning of the graph and ends at the end. This is important because it divides the applications of the tool in two: Alignment of short reads or a combined operation of aligning and merging complete sequences. This is important to point out, because we in the problem definition state that the algorithm is tuned for aligning reads shorter than the reference. As previously stated this does not obstruct the validity of the approach. It is however apparent that the tool is not optimized for these kinds of operations. We envision the approach as a part of a larger assembly process, where smaller reads are aligned, combined and eventually merged. This assembly process does not need to diverge from known the standard assembly which deals with linear references, as the indexes found in alignment are unique and can be used by overlap-techniques or as a basis for de Bruijn graphs.

Chapter 5

Validation of the approach

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

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

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

5.1 Test data

In order to avoid any ambiguity the test data in this chapter consists of small, hand-crafted sequences. As mentioned, the behaviour of the approach is determined by a relatively small set of formal instructions as to how it should behave in different circumstances. However, these instructions can be nested. This is a step which is cruical in order to produce graphs complex enough to fairly represent genetic variation. As the nesting

goes on and the structures involved grow more and more complex, finding the underlying formal rules becomes non-trivial. The subsequent tests are divided into sections, the sequences in each section are designed to represent exactly one trait of genetic data which the algorithm should handle in a specific way.

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

5.2 Tests

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

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

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

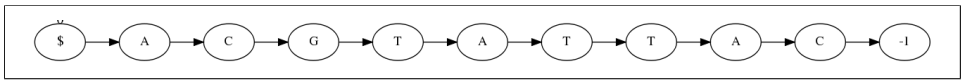


Figure 5.1: The reference graph made from the sequence "ACGTATTAC"

At last we provide a set of statements regarding the result:

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

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

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

Equal sequences

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

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

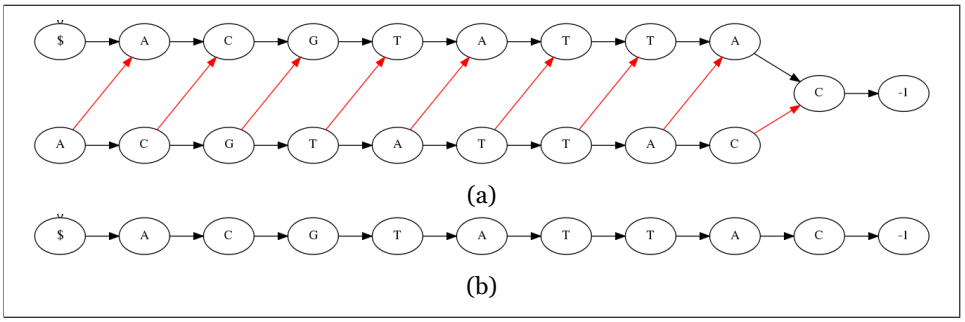


Figure 5.2: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 5.1

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

SNPs

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

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

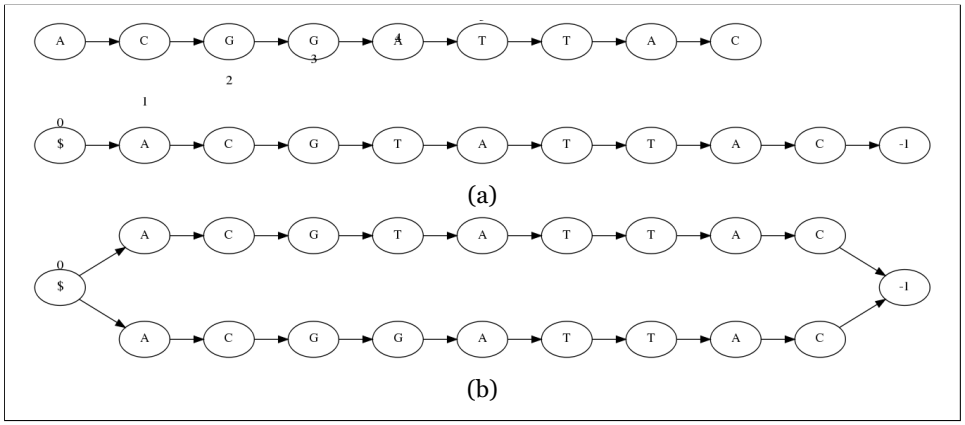


Figure 5.3: The result of aligning (a) and merging (b) the sequence "ACGTATTAC" against the reference seen in figure 5.1

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

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

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

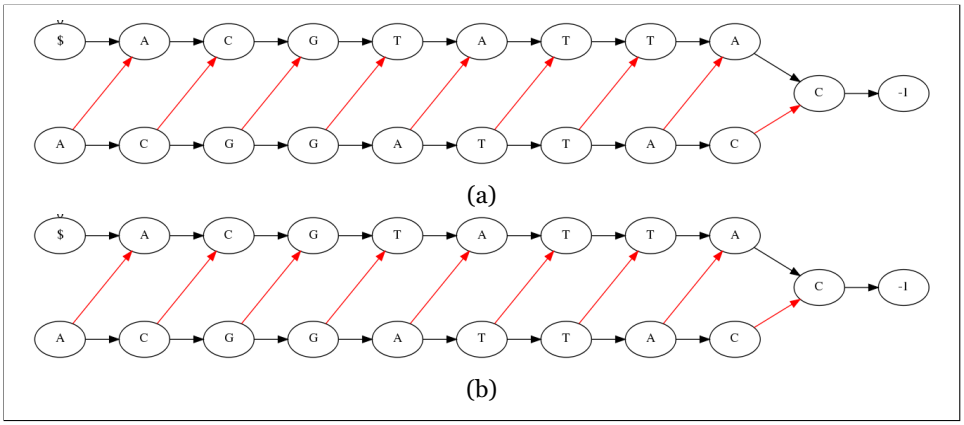


Figure 5.4: The result of aligning (a) and merging (b) the sequence "ACGGATTAC" against the reference seen in figure 5.1

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

Indels

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

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

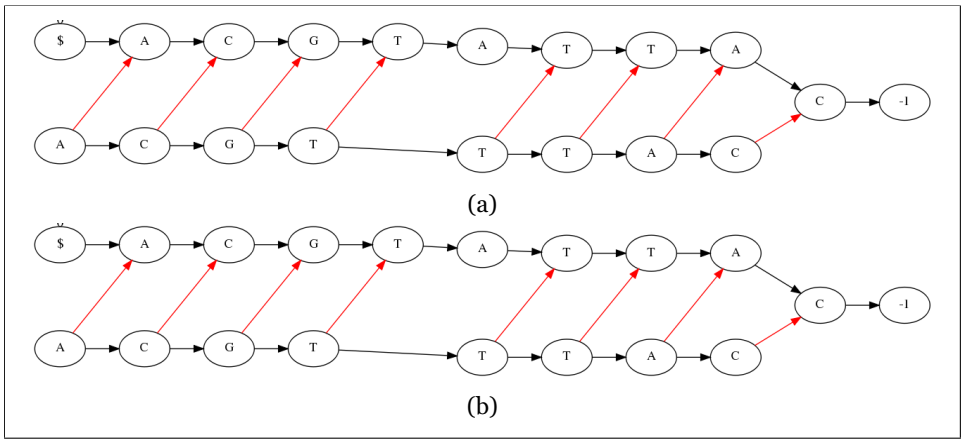


Figure 5.5: The result of aligning (a) and merging (b) the sequence "ACGTTTAC" against the reference seen in figure 5.1

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

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

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

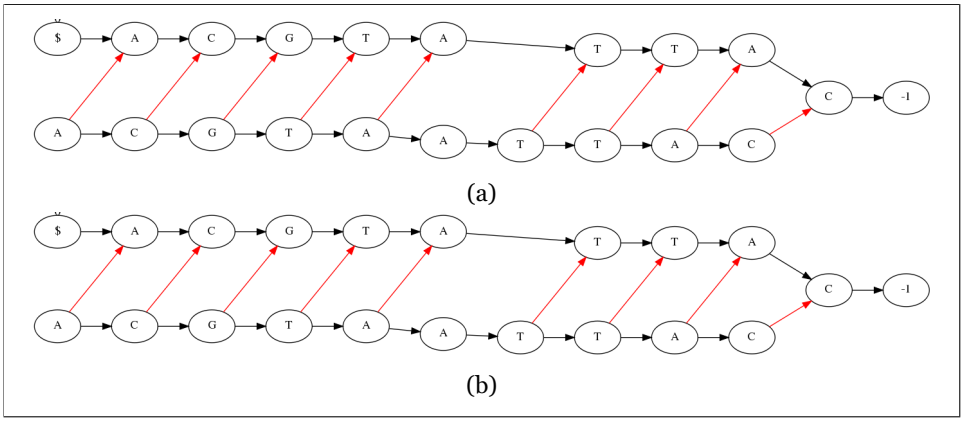


Figure 5.6: The result of aligning (a) and merging (b) the sequence "ACGTAATTAC" against the reference seen in figure 5.1

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

Structural variation

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

We will now treat structural variation as any variation which is not covered

by the point mutations shown in the previous section. Conveniently, if we drop the actual large structural changes, these mutations cover all the possible base cases of variation, and we can always define more complex cases as a combination of these. Because the number of possibilities grow exponentially we will only show a subset of cases to display the flexibilty in the approach.

We start out by combining an insertion with an SNP:

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

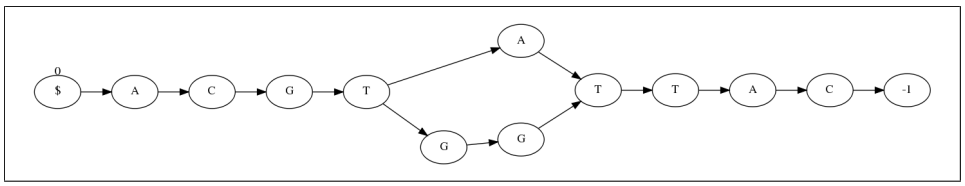


Figure 5.7: The result of merging the sequence “ACGTGGTTAC” with the reference seen in figure 5.1

We then try a number of consecutive SNPs:

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

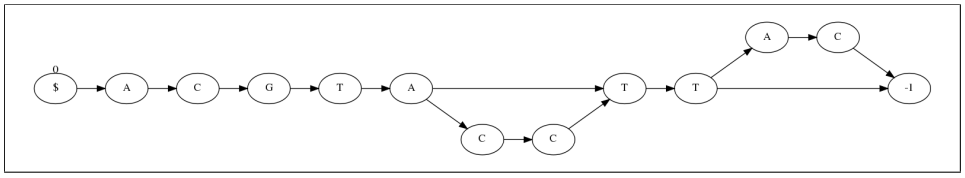


Figure 5.8: The result of merging the sequence “ACGTACCTT” with the reference seen in figure 5.1

At this point we get a result which might not be expected. Instead of four SNPs the alignment contains two deletions and two insertions. This is the previously mentioned effect of the order of operations kicking in. Importantly the somewhat suprising alignment has the exactly same alignment score as the expected one. This is also an example