UiO : Department of Informatics
University of Oslo

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Esten Høyland Leonardsen

14th March 2016

Abstract

Contents

List of Figures

List of Tables

Preface

Chapter 1

Background

1.1 Genetics

Deoxyribonucleic acid (DNA) is a molecule in which living organisms store genetic information. The information is encoded by *nucleotides* bound together by a sugar-phosphate backbone into strands. The nucleotides are smaller molecules which contain one of the nitrogenous bases ***A**denine*, ***C**ytosine*, ***G**uanine* or ***T**hymine*. Each of the bases are complementary to another, A with T and C with G. 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. DNA strands can be connected with a *reverse complementary* strand in a double helix. The two strands will have opposing directions, and every base in one of the strands will be connected to its complement. The paired nucleotides are called *base pairs*. Because either of the strands are easily deduced from the other, DNA is usually represented by only of them. DNA can be seen as a linear sequence of discrete units and can thus be represented by text strings, containing the four leading letters representing nucleotides. The text strings representations often also contain the letter N, referencing *aNy base*.

1.1.1 Gene

"What is a gene?" Helen Pearson

1.1.2 Variation

Genetic information is prone to mutations, either as a result of environmental influence or as a consequence of imperfections in reproduction. The simplest mutations are *point mutations* which affect a single nucleotide base. Point mutations can either be *Single-nucleotide polymorphisms* (SNPs) where a single base is substituted for another, or *insertions* or *deletions* (indels) where a single nucleotide is removed or inserted into the genetic sequence. Mutations can also occur over larger areas of the genome, where longer subsequences can be deleted, inserted, moved or reversed. A final type of

mutations is *Copy number variations* where a longer sequence of DNA, typically at least 1 kb [4], is repeated a variable number of times.

1.1.3 Reference genomes

1.1.4 The human genome

The human genome consists of roughly 3 billion base pairs (bp). These base pairs are spread over 22 paired chromosomes and is assumed to contain about 23 000 genes [12]. The current human reference genome is GRCh38, developed and maintained by the *Genome Reference Consortium* HOWTO: reference websites. 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.

Major Histocompatibility Complex

The *Major Histocompatibility Complex* (MHC) is a genetic region spanning approximately 4 million base pairs (mb) [7]. 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 [19]. Even more so MHC is known to be a highly variable region, containing variants that are directly associated with disease [5].

1.1.5 Sequencing

1.1.6 Alignment

Sequence alignment is the process of determining correspondence between text strings, in this case representing DNA, by mapping the elements from one to the elements of the other. The score of an alignment is determined by a *scoring schema*, which provides scores for mapping characters against characters and penalties for introducing *gaps*. A gap refers to an element in one of the strings which has no counterpart in the other string when aligned (See fig. **??**). The scoring schemas can be based around simple match/mismatch scores, which corresponds to the mathematical *Edit distance* problem, or more complex scores which models the probabilities behind the physical processes responsible for change.

1.2 Sequence graphs

1.2.1 Representation

1.2.2 Mapping

1.2.3 Alignment

1.3 Techniques and tools

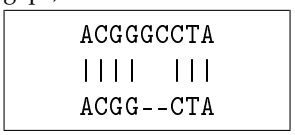
1.3.1 Dynamic programming

1.3.2 Implementing graphs

1.3.3 Suffix trees

1.3.4 Visualization of graphs

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



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

Figure 1.1: Examples of aligned text strings

Chapter 2

Implementation

In this section we will present the algorithm "Fuzzy context-based search" as an approach for mapping text strings against graph-based reference genomes. First the elements involved will be described in order to precisely define the problem. Then the syntax of the reference graphs used are explained in more detail. Finally, the algorithm is presented both through a conceptual overview and in specific detail. The implementation details refers to the tool *Graph Genome Alignment* (See Supplementary XXX).

2.1 Definitions

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 of G.

Definition 2 (Vertice)
A pair $v = \{b, i\}$ where $b \in \{A, C, T, G\}$ and i is a unique index. Every graph G also has two special vertices $s_G = \{s, 0\}$ and $t_G = \{e, -1\}$ which represents unique start and end vertices. The notation $b(v_i)$ references the first element in the pair (the nucleotide).

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

Definition 4 (Complete Path)
An ordered list P of indexes such that for all consecutive ordered pairs $\{i_x, i_{x+1}\} \in P$ there exists an edge $e = \{i_x, i_{x+1}\}$.

Definition 5 (Path)
An ordered list L of indexes such that for all consecutive ordered pairs $\{i_x, i_{x+1}\} \in L$ there exists a complete path P which starts at $\{i_x\}$ and ends at $i_{x+1}\}$.

Definition 6 (Input sequence)
A string s over the alphabet $\{A, C, T, G\}$. The length of the string is given by $|s|$. An individual character on position x is referenced by s_x

Definition 7 (Mapping score)
A score produced by mapping two characters $c_1, c_2 \in \{A, C, G, T\}$ *against a scoring matrix*

Definition 8 (Path score)
A score produced by traversing a path P *through a graph* G *to create a linear sequence, scoring gaps according to the gap penalties given by a scoring schema.*

Definition 9 (Alignment)
Given a sequence s *and a graph* G*, an ordered list* A *of indexes such that every* $a_x \in A$ *is either a valid index for a vertice in* G *or* 0*.* 0 *indicates an unmapped element of the input sequence*

Definition 10 (Alignment score)
Given a sequence s*, a graph* G *and an alignment* A*, the score produced by combining mapping scores for the pairs* $\{a_x, s_x\} for 0 <= x < |s|$ *with the path score for the path(s) provided by* A *aligned against both* G *and* s*.*

Definition 11 (The optimal alignment score problem)
For any pair $\{G, s\}$*, where* G *is a graph and* s *is a sequence, find the alignment* A *which produces the highest possible alignment score. If multiple max scores: Provide all or chose one?*

Definition 12 (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, iff the score for* A *is higher than* T*. If no such alignment exists,* s *is unmappable.*

2.2 The graph

As defined the graphs involved will be graphs where one vertice represents a single nucleotide. Every vertice also contains an identifying index, which maps uniquely to that vertice. A graph G is made by starting out with only the start and end vertices, and iteratively merging in new sequences. Every sequence merged into the graph has a corresponding path starting in s_G and ending in t_G. There is no correspondence the other way, meaning there can be complete paths from s_G to t_G which does not originate from a single sequence (See fig. **??**). There exists no information storing the origin of an edge and all paths are thus seen as equally probable when aligning a sequence. How the new sequences are merged is defined entirely through the alignment procedure which relies in part on the scoring threshold λ and the given scoring schema.

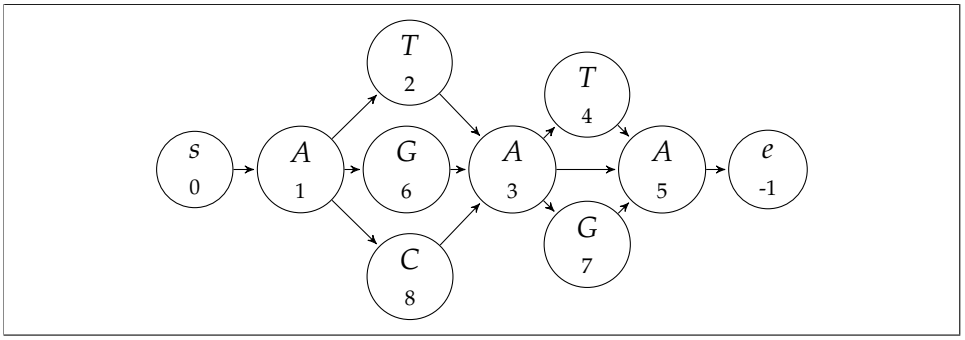


Figure 2.1: A graph made from the three sequences "ATATA", "AGAGA" and "ACAA" with 9 valid complete paths

2.3 Aligning sequences with the algorithm "Fuzzy context-based search"

2.3.1 Overview

"Fuzzy context-based search" is the algorithm we propose as a solution to the bounded optimal alignment score problem. The first step of the process is to build a searchable index based on the given graph G. This index is independent from the input sequences to be aligned, and can thus be reused for several searches. The alignment itself consists of two steps: Building a new graph G', from both G and an input string s, and search this newly formed graph for the optimal alignment. Searching for an alignment means combining nodes, representing bases, into a path which represents a linear sequence. This linear sequence can be aligned against the input sequence with regular string alignment tools and is therefore easily scorable. If the algorithm finds an alignment this is guaranteed to be one of the paths which produces the highest possible alignment score for any path in the graph (See supplementary XXX PROOF). There are some situations where the algorithm results in an empty alignment. These cases will occur when there are no paths in the graph which produces an alignment score higher than the threshold T, and the sequence s is identified as unmappable. When an empty alignment is provided as a basis for merging a new sequence into the graph, this results in a new complete path which is separated from the original vertices (See fig. 2.2)

2.3.2 Building the index

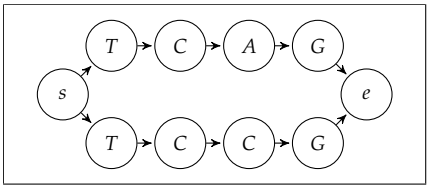
There are two data structures needed for aligning a string against the graph: a suffix tree for left contexts and a suffix tree for right contexts. Before either of the two are built the algorithm needs to decide a length for the contexts. Currently in the tool there are two ways of setting the context length: A user given parameter or an approximation based on the probability of sharing contexts Should probably ref somewhere. The length of a context does not impact the quality of the alignments found by the al-

gorithm (See Supplementary XXX PROOF) but will have an impact on the runtime (See Supplementary XXX COMPLEXITY ANALYSIS).

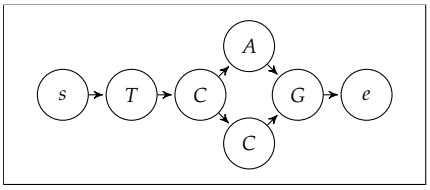
When a context length $|c|$ is set, the algorithm can start building the index. Two sets of strings, a left context set and a right context set, is generated for every vertice in the graph G. The generation of the two sets happen by the same procedure by swapping around the starting point and the direction of the iteration. When creating left contexts the algorithm starts in the start-node of G and traverses following the direction of the edges, for right contexts the opposite is done. Apart from this the two are equal. To generate the context set $c(n_x)$ for a given node n_x the algorithm looks at every string $c \in c(n_y)$ for every incoming neighbouring node n_y. Every c is modified into a new context string c' by trimming away the last character and prefixing the context with the character $b(n_y)$. All the generated strings c' is added to $c(n_x)$. As sets per definition does not allow duplicates the impact of a branching occuring in the graph will fade away after exactly $|c|$ steps as the difference is trimmed away (see Fig. 2.3), and thus avoid explosive exponentiality in the context set sizes.

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

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



(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 2.2: Different scoring thresholds T yields different reference graphs

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

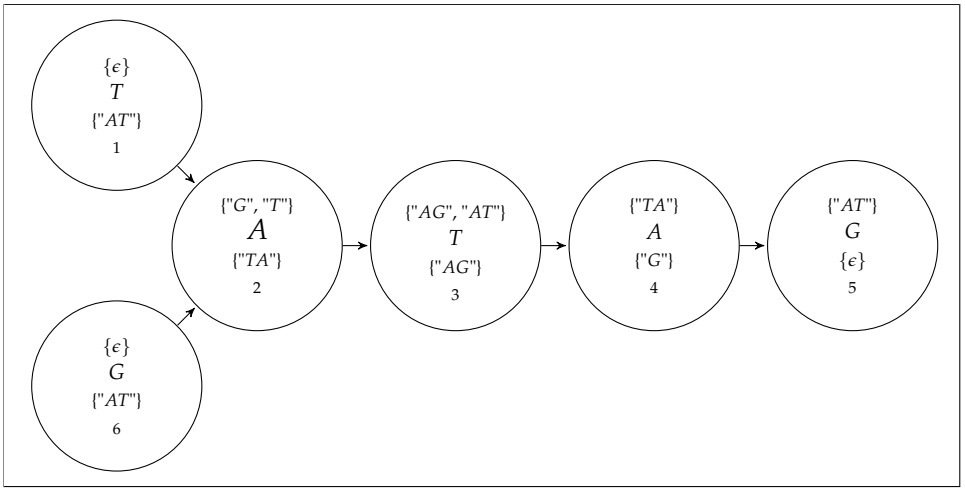


Figure 2.3: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 shown

After generating the two context sets for every node, the elements of each one is inserted into their corresponding suffix tree. Every suffix is stored as a key with the index of it's originating node as a value (fig. 2.4).In theory every node can have $4^{|c|}$ contexts in each set, in practice a more fair approximation is $b^{|c|}$ where b is the observed branching factor for the graph. Should contain something about probable values of B. Find an article on it. The current implementation uses a naive suffix tree implementation where insertion is $O(|c|)$ (Discuss more efficient suffix possibilities somewhere?), giving a total time complexity of $O(b^{|c|}|c|)$ for per node per context set and $O(2|G|b^{|c|}|c|)$ for the entire graph. Building the entire index can thus be done in $O(2|G| + 2|G|b^{|c|}|c|)$.

2.3.3 Generating the modified graph G'

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

After generating the two context sets for every node, the elements

of each one is inserted into their corresponding suffix tree. In theory every node can have $4^{|c|}$ contexts in each set. When the graph is more or less linear with few branches a more fair approximation is $B * |c|$ where B is the observed branching factor. The current implementation in the tool uses a naive suffix tree where insertion is $O(|c|)$. This is done for every node in the graph, yielding a total time complexity of $O(|G|B|c|^2)$. A discussion on more efficient suffix structures can be found in SOMEWHERE IN DISCUSSION. Every suffix is stored as a key with the index of it's originating node as a value. The total runtime for building a searchable index for a graph is $O(3|G||c|^2B)$

For every character $s_x \in s$ a left-context string and a right-context string is generated by looking at the $|c| + maxPossibleGapGivenFuzzyness(\lambda)$ surrounding characters. The two strings are treated as contexts, one left and one right, and used as a basis for a fuzzy search in it's corresponding suffix tree. The search is a recursive function based on PO-MSA. The root node is supplied with a one-dimensional scoring array corresponding to the context string c, which is initialized with all zeroes. Then, for every child, a new scoring array is computed by regular edit distance rules: For each index i take the maximal score for either a gap in the graph, a gap in the string or matching the character c_i with the character contained in the child node (Reference actual code in supplementary?), (more explanation needed?). This newly created array is supplemented to the same recursive function in the child. When a leaf node is reached the last index of the supplied scoring array corresponds to mapping the entire string c against the entire context achieved by concatenating the characters contained in the path traversed by the recursion. If the score is higher than the threshold T for the given context string, every index contained in the node is stored as a pair on the form $\{index, score\}$ in the candidate set. If an index is stored several times, only the pair containing the highest score is saved.

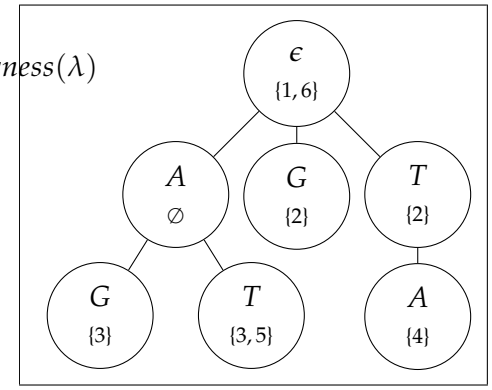
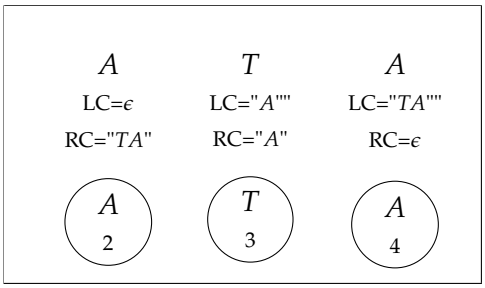
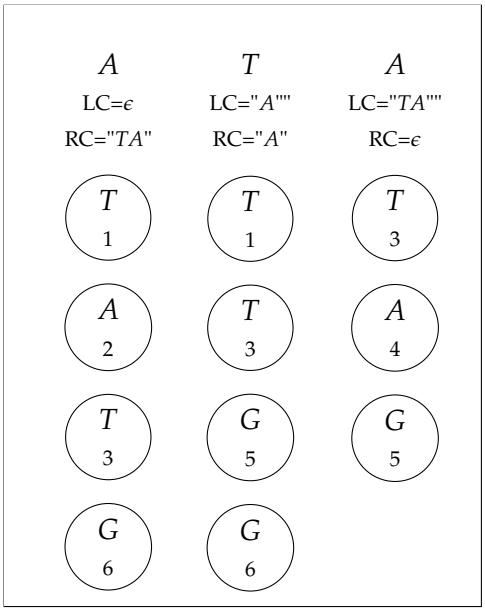


Figure 2.4: The left suffix tree corresponding to the graph in 2.3

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



(a) $T = 0$



(b) $T = 1$

Figure 2.5: The resulting candidate sets for mapping the string "ATA" against the reference genome from fig. 2.3 with varying T values

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

The newly formed graph G' can be defined formally:

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

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

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

and E' is a list of weighted edges such that

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

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

2.3.4 Searching G′ with a modified PO-MSA search

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

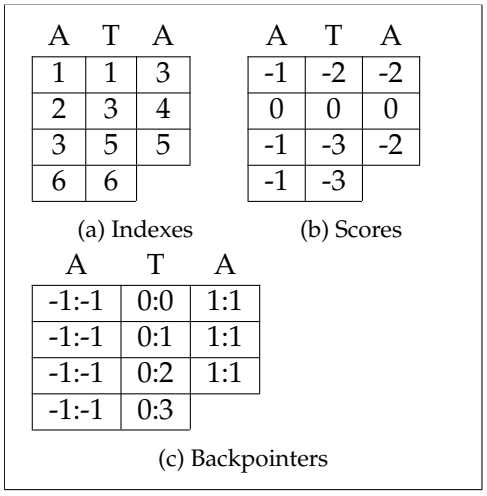


Figure 2.6: The 4 arrays used by the searching algorithm when using the candidate sets from Fig **??** and $T = 1$

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

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

Then the nodes $n_x \in V'_i$ for the remaining candidate sets at the indexes $1 <= i <= |s|$ are looped over with j as a counter, and $indexes[i][j]$ is set to x. For every such entry a list of pairs is made with other indexes (i', j') such that i' is a preceding index $i' < i$ and j' is variable looping over $indexes[i']$. Every entry-pair $((i, j), (i', j'))$ can be scored by a scoring function $\theta((i, j), (i', j'))$. The scoring function works by combining the score contained in the preceding entry, $scores[i'][j']$, a gap penalty, and a mapping score for the current index $mappingScore(n_{indexes[i][j]}, s_i)$. The gap penalty is found by combining a gap penalty for a gap of length $i - i'$ and for a gap of length

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

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

where θ is a scoring function defined as:

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

There are no restrictions in the recursive formulas to avoid alignments with aligned gaps in both the sequence and the graph. Because both gap penalties are counted the algorithm will however choose a path where only one gap is chosen in all scoring schemas where gaps are penalized with negative values, if such a path exists. By deciding the prioritized order of operations the algorithm can also handle scoring schemas with a gap penalty of 0. Within these types of schemas, the only scenario where an alignment with aligned gaps can be produced are scenarios where there are no valid candidate nodes with context scores larger than T for a subset of indexes i, which means there are no possible traversal of these nodes as a path which would give a score higher than T. This again means the path is not interesting because aligning it against the sequence would result in not mappable.
There are two components of the dynamic programming algorithm where a search is performed to find possible paths: The search backwards in the indexes and the search for distances between nodes in the graph. Both of these searches can be halted whenever the resulting gap penalty exceeds the parameter λ. This means the final time complexity is much closer related to the tunable fuzzyness parameter than the size or complexity of the graph. Because of this the time complexity for the entire dynamic programming step is $O(SOMETHINGIDIDNTPUSHTOGIT)$ (See supplementary XXX complexity analysis).

2.3.5 Handling invalid threshold values

2.4 Merging aligned sequences

Chapter 3

Method

3.1 Test data

3.2 Scoring schema

3.3 Validation

Chapter 4

Results

Chapter 5

Conclusion

Chapter 6

Discussion

Bibliography

[1] Deanna M. Church et al. 'Extending reference assembly models'. In: (2015).

[2] The 1000 Genomes Project Consortium. 'A map of human genome variation from population-scale sequencing'. In: (2010).

[3] Alexander Dilthey et al. 'Improved genome inference in the MHC using a population reference graph'. In: (2015).

[4] Jennifer L. Freeman et al. 'Copy number variation: New insights in genome diversity'. In: (2006).

[5] Roger Horton et al. 'Variation analysis and gene annotation of eight MHC haplotypes: The MHC Haplotype Project'. In: (2008).

[6] Zamin Iqbal et al. 'De novo assembly and genotyping of variants using colored de Bruijn graphs'. In: (2012).

[7] Charles A. Jr. Janeway et al. *Immunobiology: The Immune System in Health and Disease. 5th edition*. 2001.

[8] Schneeberger K. et al. 'Simultaneous alignment of short reads against multiple genomes'. In: (2009).

[9] Birte Kehr et al. 'Genome alignment with graph data structures: a comparison'. In: (2014).

[10] Christopher Lee, Cathrine Grasso and Mark F. Sharlow. 'Multiple sequence alignment using partial order graphs'. In: (2001).

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

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

[13] Shoshana Marcus, Hayan Lee and Michael Schatz. 'SplitMEM: Graphical pan-genome analysis with suffix skips'. In: (2014).

[14] Joong Chae Nal et al. 'Suffix Array of Alignment: A Practical Index for Similar Data'. In: (2013).

[15] Ngan Nguyen et al. 'Building a Pan-Genome Reference for a Population'. In: (2015).

[16] Adam Novak et al. 'Canonical, Stable, General Mapping using Context Schemes'. In: (2015).

[17] Benedict Paten, Adam Novak and David Haussler. 'Mapping to a Reference Genome Structure'. In: (2014).

[18] PA. Pevzner, H. Tang and MS. Waterman. 'An eulerian path approach to DNA fragment assembly'. In: (2001).

[19] Simone Sommer. 'The importance of immune gene variability (MHC) in evolutionary ecology and conservation'. In: (2005).