UiO : Department of Informatics
University of Oslo

Read alignment against a graph-based reference genome

Fuzzy searching in large and complex structures

Esten Høyland Leonardsen

Master's Thesis Spring 2016

Read alignment against a graph-based reference genome

Esten Høyland Leonardsen

3rd March 2016

Abstract

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

Chapter 1

Algorithm

In this section I will present the algorithm "Fuzzy context-based search" both as a conceptual idea and as a more precise implementation. The implementation details refers to the tool *Graph Genome ALignment* (See Supplementary section XXX). Before the algorithm is described the problem is defined formally. BLABLABLA

1.0.1 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. Written* $v_i = b$

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, a 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 alignment score. If multiple max scores: Provide all or chose one?

1.0.2 Overview

1.0.3 Precomputation of the graph

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 SUPPLEMENTARY?. The length of a context does not impact the quality of the alignments found by the algorithm PROOF IN SUPPLEMENTARY? but will have an impact on the runtime. A further discussion on context lengths can be found in DISCUSSION SOMEWHERE

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 node 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 stored in 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. 1.1), and thus avoid explosive exponentiality in the context set sizes.

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 $O(2|G|)$.

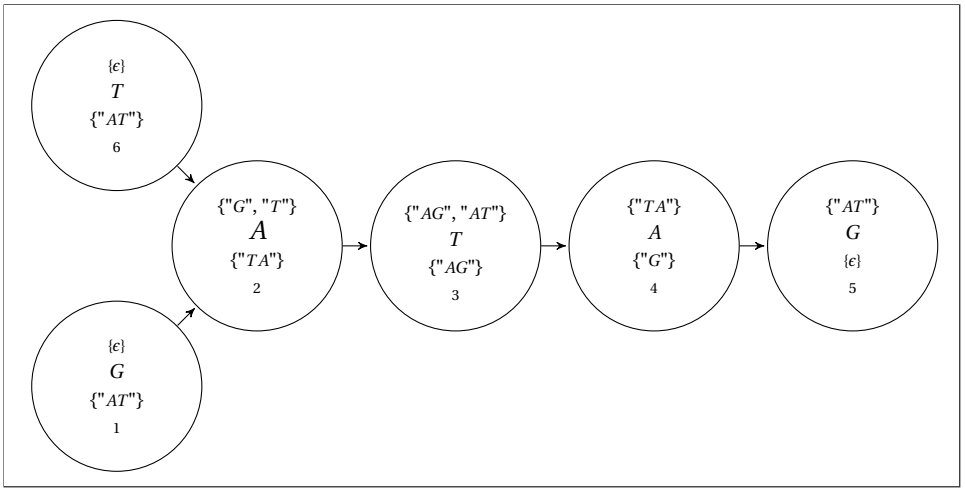


Figure 1.1: A small reference graph with left contexts (top) and right contexts (bottom) of length 2 explicitly shown

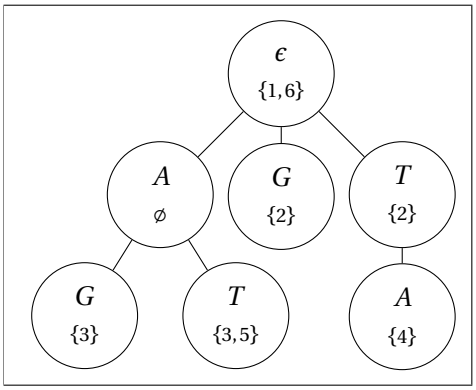


Figure 1.2: The left suffix tree corresponding to the graph in 1.1

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 $O(3|G||c|^2B)$

1.0.4 Generating the new graph G'

Creating G' is the process of determining which nodes qualifies as candidate nodes for a given input string s and combining them correctly. 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 and decides this by setting a threshold $T = maxPossibleScore(s) - \lambda$. The maximal score is found by mapping the string, be it the entire input string or a context string, against itself with a scoring function provided by the scoring schema. T is used throughout the entire process as a cutoff variable.

For every character $s_x \in s$ a left-context string and a right-context string is generated by looking at the $|c| + maxPossibleGapGivenThreshold(T)$ 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 within the threshold T the indexes contained in the leaf node is added to the set of candidates.

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

$$V'_i = \{v_x | v_x \in G \wedge contextScore(v_x, s_i) >= T\}$$

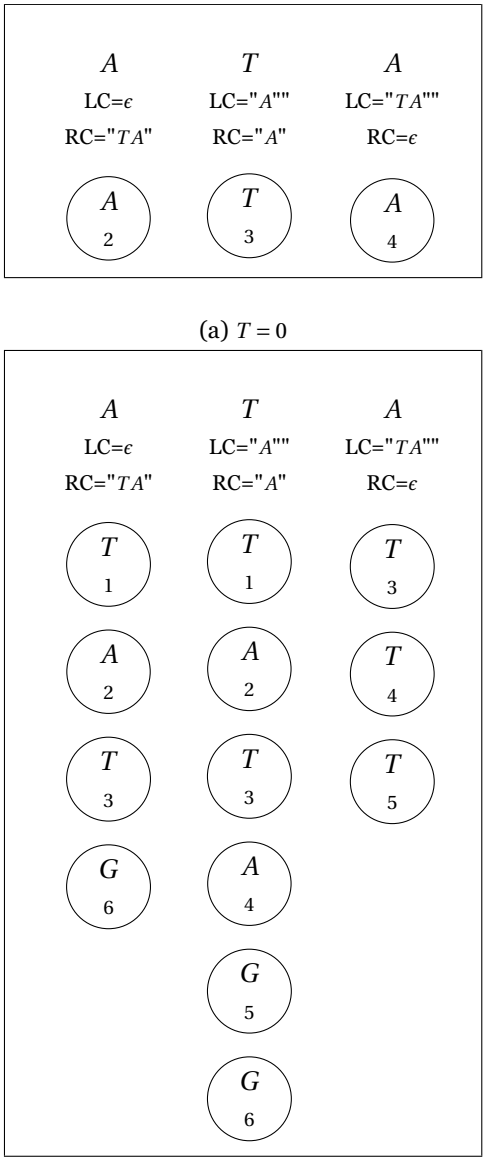
and E' is a list of weighted edges such that

$$E' = \{e' = \{i_s, i_e, w\} | i_s \in V'_x \wedge i_e \wedge V'_y \wedge$$
$$gapPenalty(y - x) <= T \wedge gapPenalty(distance(i_s, i_e)) <= T\}$$

where $contextScore(x, y)$ 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). (Something wrong in the definition of E' regarding gap penalties. Figure out what)

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*(clumsy name) falls below the threshold. 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).

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 joined by combining the scores for every node present in both sets and simply keeping the scores for nodes only present in one of the sets. During this combination operation the set is again pruned to remove all the nodes which falls under the scoring threshold when combined. The resulting set is the final candidate set for an index i. Intuitively these nodes needs to be connected to produce an actual graph. In the implementation this is done in real time by the following search algorithm by connecting every node in the candidate set for index i with every node on every other index j which can be reached without letting the gap penalty for moving from i to j exceed the threshold T. The weights of the edges are also computed in real time in the next step.



(a) $T = 0$

(b) $T = 1$

Figure 1.3: The resulting candidate sets for mapping the string "ATA" against the reference genome from 1.1 with varying T values

1.0.5 Searching G' with the modified PO-MSA

When the candidate nodes for each position has been chosen there re-