Sequence to Graph Alignment: Theory, Practice and Applications

Mikko Rautiainen

A dissertation submitted towards the degree
of Doctor of Natural Sciences
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken

Colloquium date: 25.8.2020
Dean: Prof. Dr. Thomas Schuster

Chairman: Prof. Dr. Olga Kalinina

Reviewers:
Prof. Dr. Tobias Marschall
Prof. Dr. Volkhard Helms
Prof. Dr. Knut Reinert

Scientific assistant: Dr. Pratiti Bhadra

Abstract

All species, including humans, have genetic variation between individuals. Traditionally the reference genome used for humans is one sequence that represents a mosaic of individual genomes. Recently, pangenomic approaches that take into consideration genetic diversity have become more common. One representation of pangenomes is the sequence graph or the pangenome graph, which uses a graph format to represent genetic diversity. Graphs also have other uses in bioinformatics, for example de Bruijn graphs and string graphs used for genome assembly. Due to the growing importance of sequence graphs, methods for handling graph-based data structures are becoming more important.

In this work I examine the generalization of sequence alignment to graphs. First, I present a theoretical basis for quick bit-parallel sequence-to-graph alignment. The bit-parallel method outperforms previous algorithms by a factor of 3-21x in runtime depending on graph topology. The bit-parallel method enables provably optimal sequence-to-graph alignment to scale to bacterial genomes.

Next I present GraphAligner, a practical tool for aligning sequences to graphs. GraphAligner generalizes banded alignment to graphs. Previous sequence-to-graph alignment tools could not align long reads to human sized de Bruijn graphs. GraphAligner enables sequence-to-graph alignment to scale to mammalian sized genomes. GraphAligner is as accurate as linear aligners when aligning to linear genomes. When aligning to graphs, GraphAligner is more accurate and an order of magnitude faster than previous graph alignment tools. To show the utility of GraphAligner, I present a long read genotyping pipeline, and an error correction pipeline that outperforms existing tools by a factor of two in correction accuracy and an order of magnitude in runtime.

I also show two applications where GraphAligner is an essential part. First, AERON is a tool for quantifying RNA expression and detecting gene fusion events with long reads. AERON recovered known fusion events in the K562 cancer cell line. Second, I present a hybrid graph-based genome assembly pipeline. The genome assembly pipeline uses novel methods to combine short read and long read technologies.

Kurzfassung

Jede Spezies, einschließlich des Menschen, weist genetische Variation zwischen den Individuen auf. Das Referenzgenom, das traditionell für Menschen verwendet wird, ist eine Sequenz, die ein Mosaik aus individuellen Genomen darstellt. In letzter Zeit werden pangenomische Ansätze, die die genetische Diversität berücksichtigen, immer häufiger verwendet. Der Sequenzgraph oder Pangenomgraph ist eine Methode zur Darstellung des Pangenoms, die ein Graph-Format verwendet, um genetische Vielfalt darzustellen. Graphen finden auch andere Anwendungen in der Bioinformatik, beispielsweise De Bruijn-Graphen und Stringgraphen, die für Genomassemblierung verwendet werden. Durch die wachsende Bedeutung von Sequenzgraphen werden auch Methoden für graphenbasierte Datenstrukturen immer wichtiger.

In dieser Arbeit untersuche ich die Verallgemeinerung von Sequenzalignement zu Graphen. Zuerst stelle ich die theoretischen Grundlagen für schnelles bitparalleles Sequenz-zu-Graph-Alignement vor. Die Laufzeit der bitparallelen Methode unterbietet frühere Algorithmen um einen Faktor von 3 bis 21, abhängig von der Topologie des Graphen. Darüber hinaus liefert die bitparallele Methode beweisbar optimale Ergebnisse und erreicht praxisgerechte Laufzeiten bis hin zu Eingaben der Größe eines bakteriellen Genoms, was mit existierenden Algorithmen nicht zu erreichen ist.

Danach stelle ich GraphAligner, ein praktisches Programm zur Alignement von Sequenzen an Graphen, vor. GraphAligner enthält außerdem theoretische Entdeckungen durch die Verallgemeinerung von gebändertem Alignement zu Graphen. Frühere Programme für Sequenz-zu-Graph-Alignement waren nicht in der Lage, lange Sequenzierungsfragmente an Graphen in der Größe menschlicher Genome zu alignieren. GraphAligner ermöglicht Sequenz-zu-Graph-Alignement bis zu einer Größe von Säugetiergenomen. GraphAligner ist beim Alignement an lineare Genome ebenso präzise wie herkömmliche lineare Alignier-Programme. Beim Alignement an Graphen ist GraphAligner präziser und um eine Größenordnung schneller als bisherige Programme. Um die Nützlichkeit von GraphAligner zu zeigen, stelle ich eine Genotypisierungspipeline sowie eine Fehlerkorrekturpipeline vor, die bisherige Programme um einen Faktor von 2 in der Korrekturgenauigkeit und um eine Größenordnung in der Laufzeit übertrifft.

Weiterhin zeige ich zwei Anwendungen, in denen GraphAligner eine essenzielle Rolle spielt.

Bei der ersten handelt es sich um AERON, ein Programm zur Quantifizierung von RNA-Expression und zur Erkennung von Genfusionsereignissen mittels langer Sequenzierungsfragmente. AERON konnte bekannte Genfusionsereignisse in der K562 Krebszelllinie aufdecken. Als zweites stelle ich eine graphenbasierte Methode zur Genomassemblierung vor. Die Genomassemblierungspipeline verwendet neuartige Methoden, um kurze und lange DNA-Sequenzierungstechnologien zu kombinieren.

Acknowledgments

I would like to thank my advisor Tobias Marschall for his great help during my PhD period and his flexible advising style which enabled me to pursue research in my own direction.

I would like to thank my current and previous colleagues Maryam Ghareghani, Rebecca Serra Mari, Ali Ghaffaari, Jana Ebler, Valentina Galata, Dilip Durai, Peter Ebert, Anna Feldmann, Shilpa Garg, David Porubsky, Tim Kehl and Lara Schneider for creating a positive atmosphere.

I would also like to thank my family and friends.

Contents

CHAPTER 1

Introduction

In recent years *pangenomic* approaches to explicitly represent genetic variation have become more common. One particular representation of a pangenome is the *sequence graph*, where nodes contain sequence, edges connect the nodes and paths represent known genomes. Methods to align reads to sequence graphs have existed for several years [1, 2]. However, the previous methods are impractical for aligning long sequencing reads to sequence graphs. In addition, aligning reads to de Bruijn graphs has been problematic and existing tools do not scale easily to mammalian sized graphs.

In this work I describe my work on aligning reads to sequence graphs. I present a new algorithm for aligning reads to graphs in a bit-parallel manner. In addition, I present GraphAligner, a tool that uses the algorithm in practice to align long reads to graphs, and some applications of GraphAligner. The work presented here resolves the problems of aligning long reads to genome graphs that troubled previous methods. GraphAligner enables long reads to be aligned to genome graphs accurately and quickly, and outperforms existing tools in runtime and accuracy.

This chapter summarizes previous concepts and algorithms which are used in this work or are otherwise helpful for understanding the context and importance of this work. While the ideas here could be elaborated on very deeply, the level of detail presented here roughly corresponds to how essential the concept is for understanding the content and importance of later chapters.

1.1 Genomes

The genomes of all living creatures are stored in deoxyribonucleic acid (DNA) composed of the four nucleotides adenine (A), thymine (T), cytosine (C) and guanine (G). In some creatures, like humans, the genome is organized in pairs of *homologous* chromosomes, one of which is inherited from the mother and the other from the father. Creatures with two homologous copies of each chromosome, like humans, are *diploid*. Some other creatures have only one copy of each chromosome (*haploid*) and some have more than two (*polyploid*). The sequences of the two (or more) homologous chromosomes are called *haplotypes*. When comparing two or more haplotypes within or between individuals, they will differ at some locations. The differing locations are called *variants* and the sequences are *alleles*. Variants may be classified in different ways depending on their alleles, including *single nucleotide polymorphism* (SNP) or *single nucleotide variation* (SNV), where a single nucleotide is replaced with an another one, *insertion-deletion* (indel) where one of the haplotypes contains sequence not present in the other, *copy number variation* (CNV) where a repeating sequence has a different number of copies in the two haplotypes, and *structural variation* (SV) which is often used to refer to arbitrary variations of 50 nucleotides or more.

DNA is *double-stranded*, meaning that it can be read in two opposite directions. Each double strand of DNA therefore encodes two strings: a sequence in one strand, and its *reverse complement* in the other strand. Each nucleotide is associated with its *complement*: A and T with each other, and C and G with each other. A nucleotide and its complement on the other strand is a *base pair*. The reverse complement of a string of DNA is the string *reversed* and with each nucleotide replaced with its *complement*.

The process of reading a genome is called *sequencing*. Current genome sequencing machines do not read a chromosome from end to end. Instead, small *fragments* or *reads* are sampled from everywhere in the genome. Each read contains a small substring of the chromosome, usually with errors. The information in the read is limited only to the sequence. In particular the origin of the sequenced read in terms of chromosome, location and strand is not known.

Computational approaches for sequence analysis treat the genome and the reads as strings. In general the same algorithms can be used for arbitrary string analysis and DNA sequence analysis. However, due to its biological nature, sequence analysis adds additional constraints over regular string analysis. The alphabet size of DNA strings is small and constant. The double-stranded nature of DNA means that reverse complements must also be considered.

1.2 Sequence alignment

DNA Sequencing technologies do not include information about the origin of the fragment in the genome. To reconstruct this information, *sequence alignment* (also called *approximate string matching*) is used. Given a query sequence and a reference sequence, the goal of sequence alignment is to find the substring in the reference that is “most similar” to the query. Similarity can be measured in multiple ways. The similarity can be evaluated with *scoring schemes* where a higher value represents a better alignment, for example BLOSUM scores [3], or with cost schemes where a higher value represents a worse alignment, for example *edit distance* [4]. In this work I will use the term alignment *score* for scoring schemes in general, and alignment *cost* when refering specifically to schemes where a higher value represents a worse alignment. One simple measure is the edit distance or *Levenshtein distance* [4], which measures the number of edit operations, meaning insertions, deletions and substitutions, required to transform one string into the other. Levenshtein distance uses *unit costs* as each edit has the same weight. More sophisticated measures use scores where some types of edits are more expensive than others [3], motivated by the probability of DNA

```
a e i - o u
- e i j o h
```

FIGURE 1.1 An alignment between the reference string “aeiou” (top) and the query string “eijoh” (bottom). Each character in the two strings is associated with a character or a gap (-) in the other string. There is a deletion at the leftmost index where the reference character “a” is not present in the query. There is one insertion in the middle where the query character “j” is not present in the reference. There is a mismatch at the rightmost index between the reference character “u” and the query character “h”. All other indices are matches. If unit costs are used, then this alignment has a cost of 3.

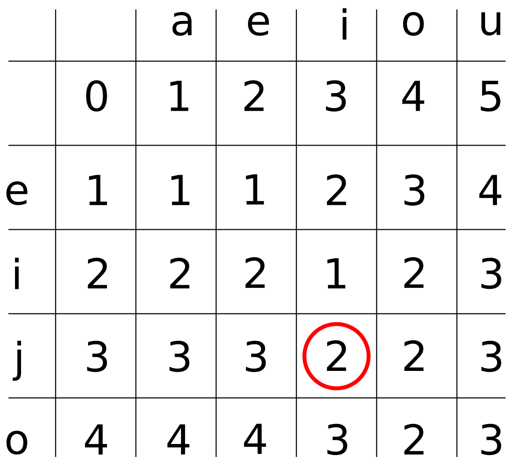
bases mutating into others.

An alignment between two strings consists of some number of *matches* where a pair of characters from the two strings are equal, *mismatches* where a pair of characters are different, *insertions* where the query sequence contains a character that the reference does not, and *deletions* where the reference contains a character that the query does not. Figure 1.1 shows an example alignment.

Sequence alignment can be performed between different parts of the strings. In *global* alignment, both strings are aligned end-to-end. In *local* alignment, any substring of the first string may be aligned to any substring of the second. In *semi-global* alignment, the entire query string must be aligned to a substring of the reference string.

Sequence alignment algorithms are typically implemented as dynamic programming (DP) algorithms. A DP table is constructed with characters from one of the sequences as rows, and characters from the other as columns. The cells of the DP table are then filled with using a recurrence on a cell’s neighbor cells. The recurrence varies depending on the algorithm used. The values of the cells represent the score of an alignment that ends at the corresponding characters in the query and the reference. Figure 1.2 shows an example DP table. The *Needleman-Wunsch* algorithm [5] was the first such algorithm, and is used for global or semi-global alignment depending on the initialization of the border cells. The *Smith-Waterman* algorithm extended Needleman-Wunsch to local alignment. The recurrence is modified to include a constant term, which represents starting an alignment at the corresponding characters in the query and the reference. Gotoh [6] extended the algorithm to *affine gap scores*, where starting a gap (insertion or deletion) has a different score than extending it. The idea is to build three matrices, one for matches and substitutions, one for insertions and one for deletions, and to modify the recurrence to represent gap starts as transitions into the appropriate matrix, and gap extensions as transitions within the appropriate matrix.

The runtime of the Needleman-Wunsch algorithm is $O(nm)$ for two sequences of length n and m. The runtime cannot be reduced to strongly sub-quadratic time, that is in time $O(n^{1-\epsilon_1}m^{1-\epsilon_2})$ for $\epsilon_1 > 0$ or $\epsilon_2 > 0$, unless the strongly exponential time hypothesis is false [7]. There are methods of calculating the DP table faster than $O(nm)$ but these methods do not reach strongly sub-quadratic time. The Four Russians algorithm [8] splits the DP matrix into chunks and uses a combination of pre-calculation and lookups to reduce runtime to $O(\frac{nm}{\log n \log m})$. Myers’ bit-parallel algorithm [9] splits the DP matrix into columns and then simulates bit-vectors using integers, calculating an entire



		a	e	i	o	u
	0	1	2	3	4	5
e	1	1	1	2	3	4
i	2	2	2	1	2	3
j	3	3	3	2	2	3
o	4	4	4	3	2	3

FIGURE 1.2 A DP table of the Needleman-Wunsch algorithm for global alignment between the strings "aeiou" and "eijo" with edit distance costs. "aeiou" is the reference on the top row. "eijo" is the query string at the leftmost column. The value at every cell is equal to the alignment cost, that is the number of edits, of the optimal alignment between the prefixes of the reference and the query at the corresponding positions. For example, the cell circled in red contains the cost of the optimal alignment between "aei" and "eij". An alignment cost of 0 represents an exact match and higher costs represent a larger number of errors.

column in one operation and reducing runtime to $O(\lceil \frac{n}{w} \rceil m)$ where w is the number of bits in a machine word. Myers' algorithm requires using edit distance as it depends on the *vertical property* which states that the difference between two vertically neighboring cells in the DP matrix is either -1, 0 or 1 [10]. Sequence alignment with arbitrary scores can be sped up by using single instruction multiple data (SIMD) instructions to compute multiple cells simultaneously, or by aligning multiple sequences simultaneously [11, 12]. Bit-parallel algorithms using arbitrary integer costs have also been described [13].

In the next sections I will first explain how the algorithms build the DP matrix, and afterwards how to recover the alignment from the DP matrix by *backtracing*.

1.2.1 Needleman-Wunsch algorithm

The *Needleman-Wunsch* algorithm [5] is the classical solution to sequence alignment. Given a cost function, the algorithm computes the optimal global alignment between two sequences. The idea is to construct a *dynamic programming* (DP) table representing the alignments of all prefixes of the two strings. The DP table is filled according to a recurrence defined by the cost function. Once the DP table has been filled, the optimal alignment can be recovered by backtracing.

The input of the algorithm is an alphabet Σ, a cost function Δ where a higher cost represents a worse match, query sequence $s = \Sigma^n$ and a reference sequence $r = \Sigma^m$. The output is an alignment between s and r with the minimal cost.

DEFINITION 1 (Recurrence for Needleman-Wunsch) Define

$$C_{i,j} = \min \begin{cases} C_{i-1,j-1} + \Delta_{i-1,j-1} \\ C_{i-1,j} + \Delta_{insertion} \\ C_{i,j-1} + \Delta_{deletion} \end{cases} \tag{1.1}$$

with the boundary condition $C_{i,0} = i\Delta_{insertion}$ for all $i \in \{1, \ldots, n\}$ and $C_{0,j} = j\Delta_{deletion}$ for all $j \in \{1, \ldots, m\}$, where $\Delta_{i,j}$ is the mismatch penalty between query character s_i and reference character r_j, $\Delta_{insertion}$ is the penalty for an insertion and $\Delta_{deletion}$ is the penalty for a deletion.

Algorithm 1 Row-wise Needleman-Wunsch algorithm

```
1: Input: a reference sequence $r$ of length $n$ and a query sequence $s$ of length $m$
2: Output: DP table $S$ representing the edit distances of the alignment of prefixes of $r$ and $s$
3: $S \leftarrow (m+1) \times (n+1)$-sized matrix of integers initialized with $S_{i,j} = i\Delta_{insertion} + j\Delta_{deletion}$
4: for $i \in [0, .., m)$ do
5:     for $j \in [0, .., n)$ do
6:         $S_{i+1,j+1} \leftarrow min(S_{i,j} + \Delta_{i,j}, S_{i,j+1} + \Delta_{insertion}, S_{i+1,j} + \Delta_{deletion})$
7:     end for
8: end for
```

The DP table can be filled *row-wise*, in which case the first row is filled left-to-right, then the second row and so on, or *column-wise* where the first column is filled top-to-bottom, then the second column and so on. Row-wise and column-wise methods produce identical output and have the same asymptotic runtime. Algorithm 1 shows the pseudocode for the row-wise Needleman-Wunsch algorithm. Note that the DP table S contains an extra dummy row and column, that is, $S_{1,1}$ represents the alignment cost of aligning the first character of s to the first character of r, and $S_{0,0}$ does not represent any alignment at all. The conversion to column-wise can be achieved by switching the order of the two loops in Lines 4 and 5. Figure 1.3 shows the DP matrix during row-wise filling.

Needleman-Wunsch may be used for semi-global alignment as well. In this case, the boundary condition is modified: instead of $C_{0,j} = j\Delta_{deletion}$ for all $j \in \{1, \ldots, m\}$, it is instead $C_{0,j} = 0$ for all $j \in \{1, \ldots, m\}$. This represents the alignment starting at any point in the reference. The backtrace must also be adjusted. Instead of taking the alignment cost from $C_{n,m}$, the alignment cost is the minimum of $C_{n,j}$ for $j \in \{0, \ldots, m\}$. The backtrace must also start at the corresponding cell. If the last row has multiple minima, there are multiple equally good alignments.

Needleman-Wunsch may also be used with scoring schemes where a higher score represents a better alignment. In this case, all minimums in the recurrence and pseudocode must be replaced by maximums. The algorithm is otherwise the same.

Implementing the Needleman-Wunsch algorithm naively would require $O(nm)$ space. However, the memory use can be improved. Hirschberg's algorithm [14] reduced space use to $O(n+m)$. The idea is to calculate the DP table from top-down and bottom-up, meeting at a cell in the middle

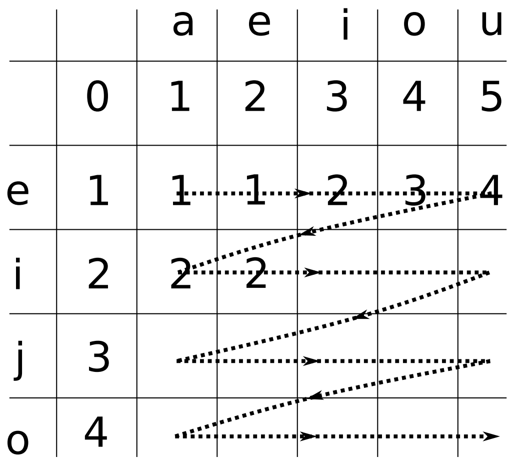


FIGURE 1.3 Row-wise filling in Needleman-Wunsch. The border cells on the top row and leftmost column are initialized with their scores and the DP matrix is filled a row at a time from left to right, top to bottom.

to discover the alignment at that position. The DP table can then be split into four parts, of which two cannot contain the optimal alignment. The procedure is then repeated in the two parts which contain the optimal alignment. Hirschberg's algorithm has also been extended to work with affine gap scores [15]. Another algorithm reduces the space usage to $O(n + \sqrt{n}m)$ [16] by using two passes, the first pass storing every $\sqrt{n}$'th row and the second pass recalculating the rows between.

1.2.2 Smith-Waterman algorithm

The *Smith-Waterman* algorithm [17] generalizes Needleman-Wunsch to local alignment. Unlike Needleman-Wunsch, Smith-Waterman algorithm requires a scoring scheme where a higher score represents a better alignment, that is it cannot be used with alignment costs. Recurrence 2 shows the recurrent for Smith-Waterman. The recurrence keeps the three terms from Needleman-Wunsch but adds one more term. The extra term is a constant 0 and represents the start of a local alignment. All local maxima in the DP table represent the end of a local alignment, and the backtrace must be modified to output all of them. The end of the backtrace can also be any cell with a value of 0.

DEFINITION 2 (Recurrence for Smith-Waterman) Define

$$C_{i,j} = \max \begin{cases} C_{i-1,j-1} + \Delta_{i-1,j-1} \\ C_{i-1,j} + \Delta_{insertion} \\ C_{i,j-1} + \Delta_{deletion} \\ 0 \end{cases} \tag{1.2}$$

with the boundary condition $C_{i,0} = 0$ for all $i \in \{1, \ldots, n\}$ and $C_{0,j} = 0$ for all $j \in \{1, \ldots, m\}$, where $\Delta_{i,j}$ is the alignment score between query character s_i and reference character r_j, $\Delta_{insertion}$

		a	e	i	o	u
	0	1	2	3	4	5
e	1	1	1	2	3	4
i	2	2	2	1	2	3
j	3	3	3	2	2	3
o	4	4	4	3	2	3

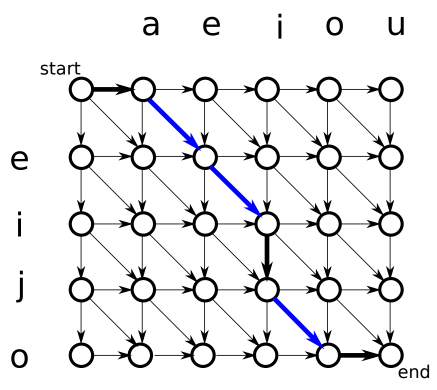


FIGURE 1.4 The relation between the DP table and the dependency graph. Left: the DP table of the global alignment between “aeiou” and “eijo”. Right: the dependency graph of the global alignment between “aeiou” and “eijo”. Each node has three directed in-edges, corresponding to the terms of Recurrence 1.1. The black edges have a cost of 1 and the blue edges have a cost of 0. Each node corresponds to a cell in the DP table and the length of the shortest path from the start node to any node is equal to the value of corresponding cell in the DP table. The optimal global alignment between the strings is equivalent to the shortest path from the start node to the end node, marked with thick edges.

is the score for an insertion and $\Delta_{deletion}$ is the score for a deletion.

1.2.3 Backtrace

After the DP matrix has been calculated, the alignment needs to be extracted from it. Some formulations of the Needleman-Wunsch algorithm record a *backtrace direction* matrix which describes the next cell in the backtrace and is calculated at the same time as the DP matrix [5]. The value in the backtrace direction matrix is set as the neighboring cell which gave the minimum value in Recurrence 1. The backtrace then starts at the last cell and follows the backtrace direction at each cell until reaching the start.

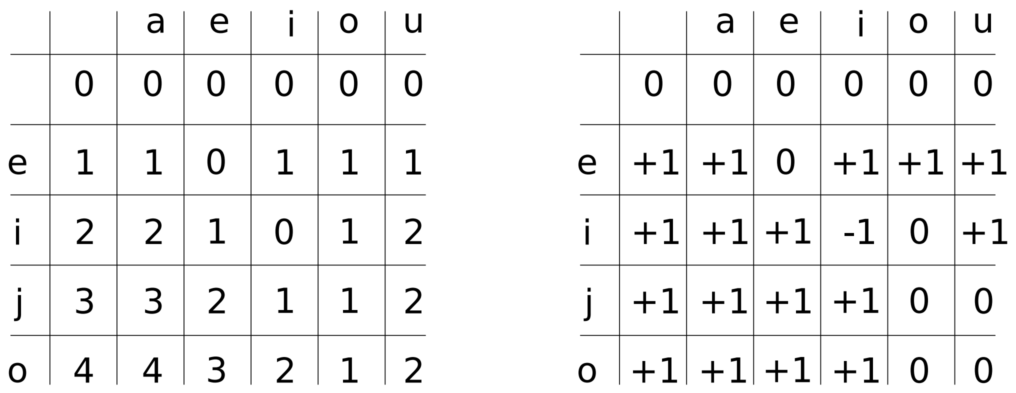
The DP table can also be considered as a *dependency graph* [18, 19], where the cells of the DP table are the nodes and the terms of the recurrence determine the in-neighbors of each cell and the weight of the edge. The alignment can then be represented as a path through the dependency graph, and the optimal alignment is the lowest weight path. Treating the DP table as a dependency graph simplifies backtracing. The dependency graph formulation allows removing the backtrace direction matrix and instead calculating the backtrace direction on the fly. Using the dependency graph formulation is also necessary for obtaining the alignment with algorithms that cannot build a backtrace direction matrix such as Myers' bit-parallel algorithm [9]. Since the dependency graph is equivalent to the DP table and the recurrence, it does not need to be explicitly stored and can be computed on the fly.

Figure 1.4 shows how the dependency graph relates to the DP table. The possible backtrace directions can be extracted from just the scores by following any in-edge to a cell $C_{i',j'}$ where the scores match, that is, $C_{i',j'} = C_{i,j} + \delta$ where δ is the weight of the edge $\{C_{i',j'}, C_{i,j}\}$ in the dependency graph. There can be multiple lowest weight paths in the dependency graph as well, representing multiple different optimal alignments.

Algorithm 2 Backtracing an optimal alignment from a DP table

```
 1: Input: a reference sequence $r = \Sigma^n$ and a query sequence $s = \Sigma^m$,
 2:  a DP table $S$ representing the edit distances of the alignment of prefixes of $r$ and $s$
 3: Output: An array $T$ representing the backtrace of an optimal alignment between $r$ and $s$,
 4:  score $c$ of the optimal alignment
 5: $i \leftarrow m$                                  ▷ Starting position of the backtrace
 6: $j \leftarrow n$
 7: $c \leftarrow S_{i,j}$
 8: while $i \neq 0 \vee j \neq 0$ do
 9:     $T.append((i, j))$
10:     if $i = 0$ then
11:         $j \leftarrow j - 1$
12:     else if $j = 0$ then
13:         $i \leftarrow i - 1$
14:     else if $S_{i,j} = S_{i-1,j-1} + \Delta_{i-1,j-1}$ then
15:         $i \leftarrow i - 1$
16:         $j \leftarrow j - 1$
17:     else if $S_{i,j} = S_{i,j-1} + \Delta_{deletion}$ then
18:         $j \leftarrow j - 1$
19:     else if $S_{i,j} = S_{i-1,j} + \Delta_{insertion}$ then
20:         $i \leftarrow i - 1$
21:     end if
22: end while
```

Algorithm 2 shows the pseudocode for finding an optimal global alignment from the DP table using the dependency graph formulation. Once the DP table is filled, the cost of the optimal alignment is retrieved from the bottom right corner cell $C_{m,n}$. The optimal alignment is retrieved by backtracing along the matrix, starting at the bottom right corner cell. The algorithm considers the DP table and recurrence as a dependency graph, and finds a path with the shortest weight from the bottom right corner to the top left corner, which is equivalent to an optimal alignment between s and r. The path is first initialized at the bottom right corner (Lines 5 and 6). Then, the path is extended by one edge at a time in Loop 8 until it reaches the top left corner. Each condition inside Loop 8 corresponds to one case in the recurrence or the initialization of the dummy row and column. Note that in the loop starting at Line 8, multiple edges may lead to a shortest weight path. In that case, there are multiple optimal alignments between the two strings. Algorithm 2 picks one of the optimal alignments in that case by prefering diagonal over horizontal edges and horizontal over vertical edges.



		a	e	i	o	u
	0	0	0	0	0	0
e	1	1	0	1	1	1
i	2	2	1	0	1	2
j	3	3	2	1	1	2
o	4	4	3	2	1	2

		a	e	i	o	u
	0	0	0	0	0	0
e	+1	+1	0	+1	+1	+1
i	+1	+1	+1	-1	0	+1
j	+1	+1	+1	+1	0	0
o	+1	+1	+1	+1	0	0

FIGURE 1.5 Left: a DP table of the semi-global alignment between the strings “aeiou” and “eijo”. The value at each row i and column j describes the edit distance of the optimal alignment between the $j-1$-length prefix of the query and any substring of the reference ending at the $i-1$’th character. Right: the same DP table using the relocatable representation. The value at each cell describes the difference between the value of the corresponding cell and the cell above in the left table.

1.2.4 Myers’ algorithm

Myers’ bit-parallel algorithm [9], also called *Myers’ algorithm* or *bit-parallel algorithm*, is an optimization of Needleman-Wunsch for faster runtime. The scoring scheme must use edit distances, where the cost of an insertion, deletion and mismatch are all 1 and the cost of a match is 0. The input and output of the algorithm are otherwise the same as Needleman-Wunsch. The algorithm exploits a special property of edit distance scoring: the difference between any two vertically or horizontally neighboring cells is in $\{-1, 0, 1\}$ [10].

Myers’ algorithm also proceeds by constructing and filling a DP table, and then backtracing over it. The difference is in how the DP matrix is represented and filled. Instead of explicitly representing the scores for each cell $C_{i,j}$, the algorithm instead uses a *relocatable* representation of $\Delta_{i,j} = C_{i,j} - C_{i-1,j}$, which describes the score difference between neighboring cells. Since there are only three possible score differences, the difference between any two vertically or horizontally neighboring cells can be represented with two bits. The score of a cell can be obtained from the relocatable representation of the DP matrix as $C_{i,j} = \Sigma_{x\in[1,..,i]}\Delta_{x,j}$. Figure 1.5 shows the relation between the DP matrix and the relocatable DP matrix.

The differences between neighboring cells can be expressed as a bit value. Myers’ algorithm uses four bits to represent the score differences between immediately adjacent cells. The bits are split on whether they represent the difference between *vertically* (V) or *horizontally* (H) adjacent cells, and on whether the value of the difference is *negative* (N) or *positive* (P). For a cell $C_{i,j}$, the values of the four bits correspond to score differences in the DP table according to the following:

$$VN = 1 \iff C_{i,j} = C_{i-1,j} - 1$$

$$VP = 1 \iff C_{i,j} = C_{i-1,j} + 1$$

$$HN = 1 \iff C_{i,j} = C_{i,j-1} - 1$$
$$HP = 1 \iff C_{i,j} = C_{i,j-1} + 1$$

In addition to the score difference bits, Myers' algorithm also requires a bit Eq describing matches in the text. Given a reference $r = \Sigma^m$ and a query $s = \Sigma^n$, the equality bit is defined with:

$$Eq = 1 \iff s_i = r_j$$

A list of *pattern bitvectors* P_σ can be precomputed from the query string for each character in $O(|\Sigma| + m)$ time and then a bitvector representing the equality bits can be retrieved based on the reference character during alignment in $O(1)$ time. The i'th bit of P_σ is set if the i'th character of s is σ, and otherwise unset.

The two vertical bit values VP and VN of each column are stored for the backtrace. The three other bits are used only as intermediate values for calculating the VP and VN bit values for the next column, and are discarded after that. In addition to these, the algorithm explicitly keeps track of the score of the last cell $C_{n,j}$ in each column. The score can be calculated from the previous column using the horizontal difference bits, that is, $C_{n,j} = C_{n,j-1} - HN_n + HP_n$. The score of the last cell must be stored for the backtrace.

Each 2×2 square of cells in the DP table can be treated as a state machine, used for computing the differences. Given the initial values for VP, VN, HP and HN, the relocatable values of the top-left, top-right and bottom-left cells are known. Then, to calculate the value of the bottom-right cell, the values for the horizontally next cell VN' and VP', and the values for the vertically next cell HP' and HN' can be calculated. Figure 1.6 shows how the bits relate to each other. To calculate VP', VN', HP' and HN', auxiliary variables Xv and Xh are defined, and the values are calculated from:

$$\begin{aligned}
Xv &= Eq \lor VN \\
VP' &= HN \lor \neg(Xv \lor HP) \\
VN' &= HP \land X_v \\
Xh &= Eq \lor HN \\
HP' &= VN \lor \neg(Xh \lor VP) \\
HN' &= VP \land Xh
\end{aligned} \tag{1.3}$$

The key insight of Myers' algorithm is that computer words of w bits can be used to simulate w-bit bitvectors, enabling the bit values to be computed in parallel for a w-cell column. Each such column can then be calculated using a constant amount of arithmetic and bitwise operations. Each word represents the values in a column as shown in Figure 1.7. Applying the equations in a cell by cell manner to calculate the relocatable DP matrix is simple. However, when attempting to calculate the DP matrix in a bit-parallel manner, there is a complication when calculating the bitvectors Xh

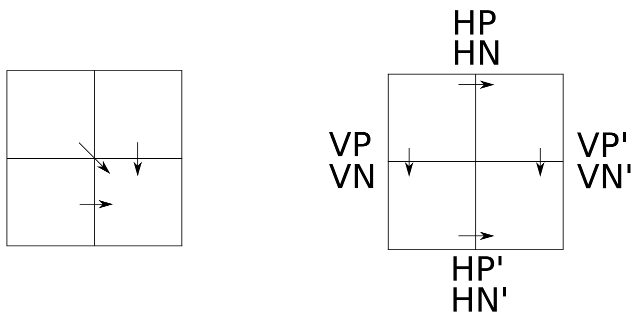


FIGURE 1.6 Left: a 2x2 part of the DP matrix. The arrows represent the terms of Recurrence 1.1 and show how the score of the bottom-right cell is defined. The values of the top-left, top-right and bottom-left cells are enough to calculate the score of the bottom-right cell, regardless of the values elsewhere in the matrix. Right: a 2x2 part of the DP matrix in relocatable representation. The arrows represent the bit values VP, VN, HP, HN. The scores of the four bits are enough to calculate the new bits VP', VN', HP', HN' regardless of the values elsewhere in the matrix, and specifically regardless of the actual alignment score in any of the four cells.

and HN, as the value of Xh in the i'th bit depends on the value of HN in the i'th bit, which in turn depends on the value of Xh in $i-1$'th bit. The computation of the bitvector for Xh therefore depends on its own value.

Myers' algorithm unwounds this circular dependency by representing Xh in terms of Eq and VP, without using HN. This proceeds by repeatedly using the identities $Xh = Eq \vee HN$ and $HN' = VP \wedge Xh$ from Equations 1.3 to replace Xh and HN. On the first iteration, this produces $Xh_i = Eq_i \vee (VP_{i-1} \wedge (Eq_{i-1} \vee HN_{i-1}))$. Continuing this, eventually we reach $Xh_i \iff \exists k \leq i, Eq_k \wedge \forall x \in [k, .., i-1], VP_x = 1$ [9]. This provides a definition of Xh which depends only on Eq and VP, which are already available. To actually calculate Xh, integer arithmetic is used. The definition states that Xh is set if there is a set bit in Eq followed by a run of ones in VP. When representing the bitvector as a binary number, the run of ones can be propagated by addition. The equation for calculating Xh is finally $Xh = (((Eq \wedge VP) + VP) \otimes VP) \vee Eq$ [9] where $\otimes$ is the exclusive or operator.

Algorithm 3 shows the pseudocode for computing the DP table using Myers' algorithm. Once the DP table has been computed, the score of any cell $C_{i,j}$ in the DP table can be computed using the bitvectors VN, VP and the explicitly stored end score $C_{n,j}$ as $C_{i,j} = C_{n,j} + \text{popcount}(VN_{i+1..n}) - \text{popcount}(VP_{i+1..n})$, where popcount is the operator for counting the number of set bits in a bitvector. Then, the backtrace proceeds as in Needleman-Wunsch.

Algorithm 3 assumes that the string is small enough to fit into a computer word, that is, $m \leq w$. When this is not the case, Myers' algorithm can be used to calculate the DP matrix such that each operation calculates a column of w cells. Lines 14- 27 provide a way of calculating VP and VN for a column of w cells, given the VP and VN values for the previous column. Myers' algorithm can then be applied either in a *column sweep* order where the first column is calculated using $\left\lceil \frac{m}{w} \right\rceil$ such operations before moving to the second column, analogously to the column-wise order of

Algorithm 3 Myers' bit-parallel algorithm [9]

```
1: Input: a reference sequence $r = \Sigma^n$ and a query sequence $s = \Sigma^m$ where $m \leq w$
2: Output: DP table representing the edit distances of the alignment of prefixes of $r$ and $s$
3:  encoded as a relocatable DP table in the arrays $VP$ and $VN$,
4:  score $c$ of the optimal alignment
5: for $i \in [0, .., |s|)$ do                                   ▷ Precompute the pattern bitvectors
6:     $P[s_i] \leftarrow P[s_i]$ OR $(1 \ll i)$
7: end for
8: $VP \leftarrow (n+1)$-sized array of integers initialized with $0$
9: $VN \leftarrow (n+1)$-sized array of integers initialized with $0$
10: $VP[0] \leftarrow 1^m$                                           ▷ $m$ ones
11: $c' \leftarrow m$
12: $c \leftarrow c'$
13: for $i \in [1, .., n]$ do
14:     $Eq \leftarrow P[s_{i-1}]$
15:     $Xv \leftarrow VN[i-1]$
16:     $Xh \leftarrow (((Eq \wedge VP[i-1]) + VP[i-1]) \otimes VP[i-1]) \vee Eq$
17:     $HP \leftarrow VN[i-1] \vee \neg(Xh \vee VP[i-1])$
18:     $HN \leftarrow VP[i-1] \wedge Xh$
19:     if $HP_m$ then
20:         $c' \leftarrow c' + 1$
21:     else if $HN_m$ then
22:         $c' \leftarrow c' - 1$
23:     end if
24:     $HP \leftarrow HP \ll 1$
25:     $HN \leftarrow HN \ll 1$
26:     $VP[i] \leftarrow HN \vee \neg(Xh \vee HP)$
27:     $VN[i] \leftarrow HP \wedge Xv$
28:     $c \leftarrow \min(c, c')$
29: end for
```

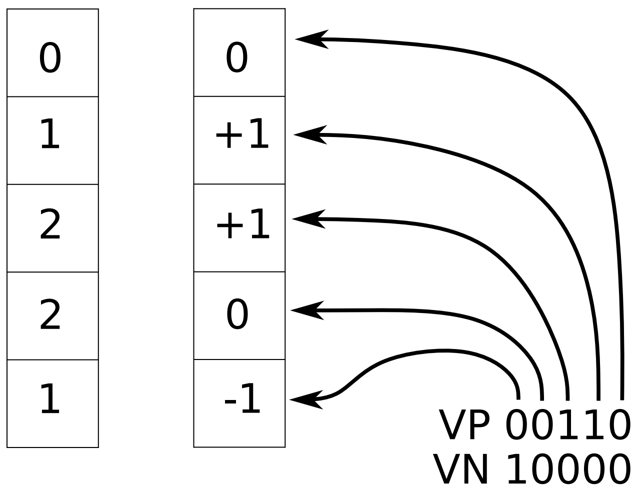


FIGURE 1.7 Encoding of the bitvectors in Myers' algorithm. Left: a column in the DP table. Middle: the same column in the relocatable representation. Right: The VP and VN bitvectors which are equivalent to the relocatable column. The arrows show which indices of the bitvectors represent which cells in the column.

Needleman-Wunsch, or in a *row sweep* order where the DP matrix is calculated w rows at a time and repeated $\left\lceil \frac{m}{w} \right\rceil$ times, analogously to the row-wise order of Needleman-Wunsch. The runtime of Myers' algorithm is $O(n \left\lceil \frac{m}{w} \right\rceil)$ in either order.

1.3 Exact matching

Exact matching refers to finding an exact match of a query string within a reference string. Exact matching between two strings can be accomplished in a bit-parallel manner with the *Shift-And* algorithm [20–23] and the closely related *Shift-Or* algorithm, described in more detail in Section 1.3.1. The runtime of Shift-And and Shift-Or is $O(\lceil \frac{n}{w} \rceil m)$ where w is the number of bits in a machine word. Exact matching between two strings can also be accomplished in optimal $O(n+m)$ time by indexing the reference with an FM-index [24] and then searching it for the query string.

1.3.1 Shift-And algorithm

The *Shift-And* algorithm [20–23] is used for exact matching between two strings. The input is a query string $s = \Sigma^m$ and a reference string $r = \Sigma^n$. The output is the end positions of exact matches between the whole query string and a substring of the reference string, equivalent to semi-global alignments with an edit distance of 0.

Shift-And may be interpreted as simulating a non-deterministic finite automaton whose accepting language is the query string, and feeding the reference string into it. It can also be interpreted as a DP table which uses bit values, each representing the end of an exact match. This section uses the DP table interpretation due to its connection with approximate string matching and in particular the Needleman-Wunsch algorithm.

Shift-And uses maching words to simulate bitvectors for representing the columns in the DP table. Using machine words enables parallel processing to check the exact matches of all prefixes of the query string simultaneously. The idea is to keep an m-length bitvector of matches per column, where a set bit represents an exact match ending at the corresponding position in the query and the reference. By using the bitvector of a previous column, the next column can be computed in constant time. The algorithm requires pre-processing the query string to build the pattern bitvectors P_σ. If the query sequence is longer than the number of bits in a machine word, longer bitvectors can be simulated by using multiple machine words, similarly to Myers' algorithm.

Algorithm 4 Shift-And algorithm

```
 1: Input: a reference sequence $r = \Sigma^n$ and a query sequence $s = \Sigma^m$ where $m \leq w$
 2: Output: DP table $S$ representing exact matches between prefixes of $r$ and $s$
 3: $P \leftarrow |\Sigma|$-sized array of integers initialized with $0$
 4: **for** $i \in [0, .., m)$ **do**                    ▷ Precompute the pattern bitvectors $P$
 5:     $P[s_i] \leftarrow P[s_i]$ OR $(1 \ll i)$
 6: **end for**
 7: $S \leftarrow n$-sized array of integers initialized with $S[i] = 0$
 8: $S[0] \leftarrow 1$ if $s_0 = r_0$ and $0$ otherwise
 9: **for** $i \in [1, .., n)$ **do**
10:     $S[i] \leftarrow ((S[i-1] \ll 1) + 1)$ AND $P[r_i]$
11: **end for**
```

Algorithm 4 shows the pseudocode for Shift-And. In each column, the previous word is first shifted left by one bit, and then logical AND-ed with the character bitvector of the reference character, giving the algorithm its name. Shifting the word first extends the matches by one character, the $+1$ initializes a new match, and AND-ing filters out positions which do not match the current character in the reference. Once the DP table has been computed, the last bit of a column in the DP table is set if and only if there is an exact match ending at the corresponding position in the reference. The algorithm assumes that the number of bits in an integer is at least as many as the length of the query sequence s. For w-bit machine words, when $m \leq w$, this is trivially true, and otherwise larger integers may be simulated by using $\lceil \frac{m}{w} \rceil$ machine words. The runtime of Shift-And is $O(\lceil \frac{m}{w} \rceil n)$. Normally the output of Shift-And is only the match positions and not the DP table, and the DP table is not even explicitly stored, instead only the rightmost column is kept during calculation.

A variant of the Shift-And algorithm is the *Shift-Or* algorithm, which instead uses unset bits to represent matches. Shift-Or is similar to Algorithm 4, except the 0 in Line 3 is replaced with 1^m (m ones), 0 and 1 are switched around in Lines 5, 7 and 8, the OR in Line 5 is replaced with AND, the AND in Line 10 is replaced with OR, and the $+1$ is removed from Line 10. The benefit of Shift-Or is using one fewer operation per column when calculating the matrix in Line 10.

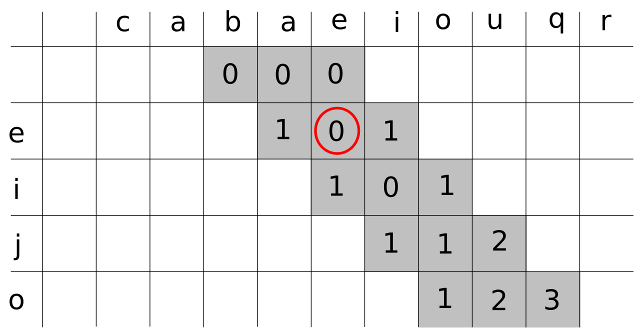


FIGURE 1.8 Banded alignment. Given the information that the optimal alignment starts at the cell circled in red with a match, a parallelogram (cells with a gray background) can be drawn around the start. Only cells inside the parallelogram are calculated. Cells outside of the parallelogram (cells with a white background) are not calculated.

1.4 Banded alignment

Banded alignment [19] refers to only calculating a part of the DP matrix instead of the entire matrix. In theory, calculating just the cells involved in the optimal alignment would suffice, but that is not practical as it would require knowing where the optimal alignment is. Given a known alignment starting position, banded alignment commonly designates a specific diagonal parallelogram of the DP matrix as the band, whose width is determined by a *banding parameter* [19, 25]. Figure 1.8 illustrates this. A higher banding parameter leads to a higher probability that the parallelogram contains the optimal alignment and higher runtime. Given a banding parameter b, the runtime only depends on the length of the query and the banding parameter, with no dependence on the length of the reference. The optimal alignment is guaranteed to be found if the number of errors is less than b. The band can also be dynamically moved instead of being pre-determined at the start of computation [26]. Since banded alignment only affects which parts of the DP table are calculated, it can be combined with any method of calculating the DP table. Virtually all alignment programs use banded alignment in practice [27–30].

1.5 Indexing

Indexing a string means building a data structure over the string which supports finding exact substrings efficiently. Some data structures enable arbitrary substring queries, where the query string is of arbitrary length, while others support only constant length queries.

Suffix trees [31] and *suffix arrays* [32] are data structures which support querying arbitrary length strings. Suffix trees and arrays can be constructed from a string of length n in linear time [33, 34]. *Enhanced suffix arrays* [35] supplement suffix arrays with auxiliary information. Suffix trees and enhanced suffix arrays can run the same operations in the same runtime and enable constructing one from the other, so they are in a sense equivalent [35]. Although the asymptotic runtimes of suffix trees and enhanced suffix arrays are identical, enhanced suffix arrays are preferred in practice due to using less memory.

Burrows-Wheeler transform [36] (BWT) is a related method for reordering a string by shuffling its characters. BWT transforms a string by ordering its suffixes and extracting the character before the suffix in the order. The BWT of a string can be constructed from a suffix array. BWT does not directly enable querying the string, but the *FM-index* [24] can be used to index a BWT-transformed string. The FM-index has a memory advantage over suffix arrays, as the memory used to index a string of length n with alphabet size σ is $O(n \log \sigma)$, while suffix arrays require $O(n \log n)$ space.

K-mer indices [37] were some of the first indexing methods used in bioinformatics software. In a k-mer index, substrings of length k (*k-mers*) along with their positions in the reference are stored. Then, to align a query, k-mers are extracted from the query and retrieved from the index. This provides a list of exact matches of length k between the query and the sequence. Some early k-mer indices [37] flipped the role of the reference and the query during indexing, that is, the index is built over the sequence to be aligned and the entire database is scanned for matching, due to memory concerns. A k-mer index may store either all k-mers in the reference, or some subset of k-mers, for example every n'th kmer.

A specific kind of a k-mer index is the *minimizer index*. The *minimizer* of a set of k-mers is the smallest k-mer under a given ordering. Minimizers are a form of *locality sensitive hashing* [38], meaning that the probability that two sets have the same minimizer is proportional to the similarity between the sets. When applied globally to a string, this enables finding similar sequences quickly [39]. *Minimizer winnowing* [40, 41] selects a subset of k-mers for indexing. A *window* of size w is slid through the reference. Each window contains w k-mers, and the smallest k-mer in each window is included in the index. This samples the k-mers sparsely instead of storing all k-mers, and provides a consistent method of selecting which k-mers get selected. If two sequences have an exact overlap of $w + k$ characters, they also share a k-mer sampled by minimizer winnowing. The performance of minimizer winnowing, measured by the density of sampled k-mers, depends on the k-mer order used [42].

Several alignment softwares use suffix trees [43], suffix arrays [27], FM-indices [28, 44–46], k-mer indices [30] and minimizer indices [29, 47, 48].

1.6 Match chaining

Indices can be queried to find exact matches, or *seed hits*, between two strings. The exact matches can then be *chained* to produce an approximate alignment between strings. The sequences between the chained matches are then aligned using an exact DP alignment algorithm. This is the *seed-and-extend* paradigm used by many aligners [27–30].

The goal of match chaining is to find the approximate region where the read will align. Given a list of seeds $(q_1, r_1, l_1), ..., (q_n, r_n, l_n)$ where q is the position in the query, r is the position in the reference and l is the length of the match, match chaining algorithms will select a subset of seeds which maximizes some particular scoring function. Match chaining can be solved with dynamic programming, which finds the optimal solution in $O(n \log n)$ time using a range minimum query data structure [49, 50]. Match chaining can also be done heuristically by only considering a subset of possible extensions [29].

Seeds can also be *clustered* [30, 47] by considering seeds on similar *diagonals*. The diagonal of a seed (q_i, r_i, l_i) is defined as $r_i - q_i$. Seeds which are part of the same alignment will be in nearby diagonals, although seeds which are on nearby diagonal are not necessarily part of the same alignment and can be incompatible with each other. For example seeds (q_i, r_i, l_i) and $(q_i + 1, r_i, l_i)$ will be in adjacent diagonals but they cannot be a part of the same alignment.

1.7 Pangenomes and genome graphs

Human genomes contain variation between individuals [51–55]. Recent projects have found large amounts of novel sequence not contained in the reference genome [56, 57]. Genetic diversity in humans is implicated in many phenotypic variations such as gene expression and susceptibility to diseases [53, 58, 59].

Given the amount and importance of genetic variation between humans, it is important to properly take variation into account when performing genetic analyses. Traditionally the reference genome has been one sequence that does not correspond to any specific individual's genome. Recent methods have considered *pangenomic*[1] approaches that aim to preserve genetic variation between individuals [60].

There are multiple ways of implementing pangenomic data structures. The commonly used reference genomes by the Genome Reference Consortium (GRC) include variable genomic regions as alternate alleles [61]. Some methods have represented a pangenome simply as a collection of genomes [62]. Journaled string trees [63] is a method for processing multiple genomes which allows sequential algorithms to process the common sequences only once. Haplotype panels [64] represent a population of genomes as a matrix with variant sites as columns and haplotypes as rows, marking the variant of each haplotype in the cells.

Genome graphs [65], also called *sequence graphs* [1] and *pangenome graphs* [60], represent a collection of sequences in a graph structure. The sequence is stored in the nodes of the graphs and the edges mark connections between the sequences. A genome or other sequence can then be represented as a path in the graph. Graphs can collapse the common substrings of the sequences into a shared part that is only represented once. This has technical advantages in reducing storage requirements, and interpretive advantages in separating the sequences into shared and unique regions.

Genome graphs are commonly *bidirected* [65], meaning that each node has two distinct ends

[1]sometimes spelled "pan-genomic" in older publications

and the edges connect ends of nodes. Nodes can then be traversed in forward direction (left-to-right) with the node label or backward direction (right-to-left) with the reverse complement of the node label. This represents the double stranded nature of DNA. A walk in a bidirected graph must enter a node from one end, and then leave it from the opposite end. A specific type of genome graph is the *variation graph* [1] which is a bidirected graph with no overlap between nodes.

De Bruijn graphs [66] are a type of graph used in genome assembly. Although they are used for assembling one genome, they are technically similar to pangenomic graphs and the same techniques apply to handling them. De Bruijn graphs are composed of *k-mers*, which are strings of length k. Two nodes are connected by an edge if they share a $k-1$-substring as a prefix or suffix, and the edge is labeled by the shared $k-1$-length substring [67]. De Bruijn graphs have also been formulated [66] with edges labeled by k-length strings, and nodes with $k-1$ length strings, where an edge connects to a node if its $k-1$-prefix (or suffix) is equal to the node label. De Bruijn graphs are commonly *compacted*, where non-branching paths or *unitigs* are collapsed into one node. In this work I consider bidirected compacted de Bruijn graphs with k-length nodes and $k-1$-length edges as the existing tools [67, 68] support this formulation. De Bruijn graphs can further be extended to explicitly represent a pangenome as a *colored de Bruijn graph* [69–71], where each k-mer can have one or more color labels. The labels can represent for example different genomes, and colored de Bruijn graphs can efficiently be queried to find which genomes contain a specific k-mer or set of k-mers [72].

1.7.1 Superbubbles

A *superbubble* [73] is a structure in a graph consisting of a start node, an end node and a set of contained nodes. Every path from the start node must lead to the end node, and every path reaching the end node must pass through the start node. The nodes within those paths are called the contained nodes. A superbubble must contain no edges from an internal node to a node outside of the superbubble, or edges from outside the superbubble to an internal node. The superbubble must also contain no cycles, that is, the induced graph of the contained nodes must be acyclic. A node by itself is also a superbubble where the start and end nodes are the same and the set of contained nodes is just the node itself. Superbubbles are defined only by the graph topology and the node labels are irrelevant. Superbubbles in a sequence graph can represent genetic variation [74]. Genetic variation will also usually result in bubbles in a de Bruijn graph. Superbubbles with two paths and four nodes arranged in a diamond-like shape (Figure 1.9) are called *simple bubbles*. Simple bubbles usually arise from SNPs in a genome, but other kinds of variation can also introduce simple bubbles.

A *chain* of superbubbles is an acyclic subgraph with one start superbubble, one end superbubble and a set of contained superbubbles. The end node of each superbubble, except the last superbubble, must also be the start node of an another superbubble in the chain. Similarly, all start nodes, except the first superbubble's start node, must be end nodes of an another superbubble. A superbubble by itself is also a chain of superbubbles. Any path through a chain of superbubbles must pass through all start and end nodes of the superbubbles. A chain of superbubbles can be thought of as an “almost linear” subgraph. Figure 1.9 shows an example of a chain of superbubbles.

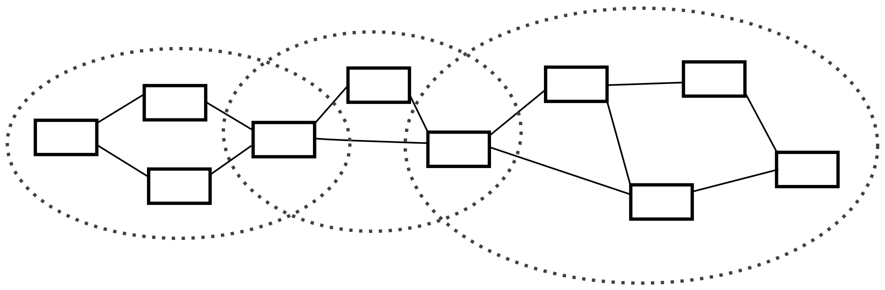


FIGURE 1.9 A chain of superbubbles. The rectangles are nodes, and the solid lines connecting them are edges. The chain is composed of three superbubbles. The dashed grey circles show the three superbubbles. There is one simple bubble on the left, a superbubble involving an indel in the middle and a complex superbubble on the right.

1.8 Sequence-to-graph alignment

Aligning sequences to a genome graph is a fundamental operation used in analyses such as genotyping [75] and error correction [2, 76]. Given a graph and a query string, the sequence-to-graph alignment problem refers to finding a path in the graph such that the alignment cost between the query string and the labels of the path is minimized. The first algorithms for aligning reads to a graph were discovered outside bioinformatics. In 1989, an algorithm for approximately matching a string to a regular expression introduced by Myers [77]. Since regular expressions are equivalent to deterministic finite automata, and genome graphs can be represtented as directed graph and therefore convertible to a deterministic finite automaton, this algorithm could in principle be used for aligning sequences to a graph with errors. The runtime of the algorithm is $O(mn)$ where m is the length of the string and n the length of the regular expression. In 2000, Navarro [78] discovered an algorithm for aligning strings to graphs in the context of hypertext searching. The algorithm is a generalization of the well known Needleman-Wunsch [5] algorithm to directed graphs and runs in $O(|V| + m|E|)$ time where m is the length of the string to be aligned. The algorithm processes the DP matrix in a row-wise manner, passing each row in the dynamic programming matrix twice; first to set the “diagonal” and “vertical” terms of the recurrence, and then to set the “horizontal” terms, using a depth first search to propagate the horizontal terms, described in more detail in Section 1.8.1. It was later proven [79] that this runtime is in fact optimal unless the strongly exponential time hypothesis is false. The runtime is equivalent to linear alignment when the graph is a linear string of nodes. An interesting note is that while exact matching between strings can be accomplished in linear time, faster than approximate matching, exact matching between a sequence and a graph has the same lower bound as approximate matching [79].

Other alignment algorithms have been discovered in bioinformatics. Although published later than Navarro's algorithm, these methods do not have optimal runtime or do not apply to general graphs. Partial order alignment (POA) [80] generalizes Needleman-Wunsch to directed acyclic graphs (DAGs). POA is a special case of Navarro's algorithm where the graph is acyclic. A cyclic graph can be "unrolled" into a DAG [1], which enables using POA on the unrolled graph. This is the approach taken by the vg toolkit [1, 65]. V-align [81] aligns sequences to general graphs in time $O(m|E||V' + 1|)$ where $|V'|$ is the size of the graph's feedback vertex set. BGreat [82] aligns reads to de Bruijn graphs in an approximate manner that is not guaranteed to find the optimal alignment. HybridSPAdes [83] uses the dependency graph formulation and applies Dijkstra's algorithm to find the lowest weight path in the DP matrix, which takes $O(|E|m + |V|m \log(|V|m))$ time. LoRDEC [2] uses a depth-first traversal to align to a de Bruijn graph, which can take exponential time in the worst case. Minigraph [84] aligns reads approximately using a seed-and-chain strategy, which might not align to variant-dense regions.

In recent years, the field of bioinformatics has rediscovered Navarro's algorithm and reached asymptotically optimal runtimes. Affine gap costs can be generalized to graphs by including extra subgraphs for insertions and deletions [85], similar to linear alignment. Navarro's algorithm has further been generalized to arbitrary match costs [86] in $O(V + mE + \log n)$ runtime where n is the number of possible match costs. Since n is constant in practice, this keeps the optimal asymptotic runtime. SPAligner [87] is the sequence-to-graph aligner used in the SPAdes assembler toolkit, using Dijkstra's algorithm to find the lowest weight path in the DP matrix in optimal time when using Levenshtein distance. The authors note that the arbitrary match costs algorithm [86] could be combined with the Dijkstra-based approach to achieve optimal runtime with arbitrary match costs.

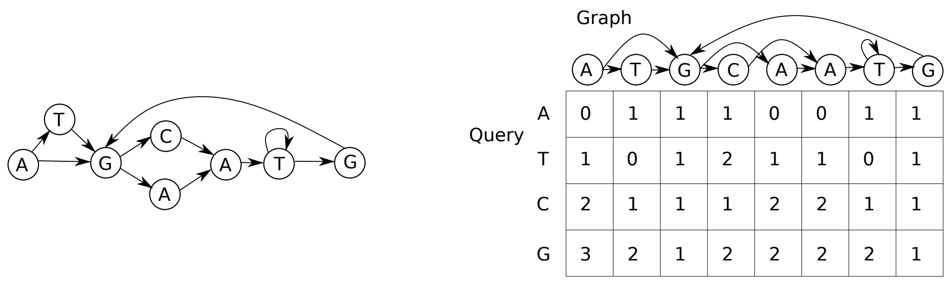
1.8.1 Navarro's algorithm

Navarro's algorithm [78] generalizes the Needleman-Wunsch algorithm [5] to directed graphs. Instead of aligning a query sequence to a reference sequence, Navarro's algorithm aligns a query sequence to a reference graph. In contrast to the Needleman-Wunsch algorithm, where the DP table can be filled in a row-wise or column-wise manner, Navarro's algorithm requires filling the DP table in a row-wise manner.

The input of Navarro's algorithm is an alphabet Σ, query sequence $s = \Sigma^m$, and a reference graph $G = (V, E \subseteq (V \times V), \sigma : V \to \Sigma)$, where V is the node set, E is a set of directed edges and σ is a function that appoints one character to each node as the node label. The output is a DP table indirectly representing the possible alignments between the query and the reference.

We notate the in-neighbors of a node as $\delta_v^{in} = \{x : (x, v) \in E\}$ and similarly the out-neighbors as $\delta_v^{out} = \{x : (v, x) \in E\}$.

Given a sequence graph (V, E, σ) and a query sequence $s = \Sigma^m$, the algorithm fills a DP table C of size $m \times |V|$ where rows are indexed by integers representing base pairs in the sequence, and columns are indexed by nodes in the graph. Figure 1.10 shows how the DP table is connected to the



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

FIGURE 1.10 The DP table in Navarro's algorithm. Left: an input graph. Right: The DP table of aligning the string "ATCG" to the graph from left. Each row corresponds to a character in the query sequence, similar to linear alignment. Each column corresponds to one node in the graph. The nodes may be ordered arbitrarily and the output and asymptotic runtime are not affected. The score of a cell is the best score of an alignment ending at the corresponding character in the query sequence and node in the graph.

graph. The ordering of the columns does not affect the asymptotic runtime or the correctness of the algorithm. The value of a cell in the DP table represents the edit distance of an alignment ending at the corresponding location in the query and the graph. The recurrence used to calculate the values is generalized from the recurrence in the Needleman-Wunsch algorithm. Instead of depending on the values of the neighboring column, the recurrence depends on the values of the in-neighbor columns as defined by the graph topology.

DEFINITION 3 (Recurrence for SGA) Define

$$C_{i,v} = \min \begin{cases} C_{i-1,u} + \Delta_{i,v}, & \text{for } u \in \delta_v^{in} \\ C_{i,u} + 1, & \text{for } u \in \delta_v^{in} \\ C_{i-1,v} + 1 \end{cases} \tag{1.4}$$

with the boundary condition $C_{0,v} = \Delta_{0,v}$ for all $v \in V$, where $\Delta_{i,v}$ is the mismatch penalty between node character $\sigma(v \in V)$ and sequence character s_i, which is 0 for a match and 1 for a mismatch.

The three terms in the recurrence can be split into "diagonal" (top), "horizontal" (middle) and "vertical" (bottom), named after their relative positions in the DP table. The recurrence can have cyclic dependencies where the value of a cell in the matrix depends on itself. However, for any graph and any sequence, there is exactly one unique solution which satisfies Recurrence 1.4 [85]. The key insight in Navarro's algorithm that after using the diagonal and vertical terms to fill a row, the calculated values are at most one higher than the optimal values. The horizontal term can then be used to correct the values in the row to the optimal values by propagating it with a depth first search. Algorithm 5 shows the pseudocode for Navarro's algorithm. The algorithm works in two passes: first, iterate over all cells to calculate the diagonal and vertical terms (Lines 16-21); then, iterate over all cells to find cells which can be updated by the horizontal term, and using a depth-first search to recursively update all out-neighbors of the cell (Lines 22-26). The first pass clearly runs in $O(|V| + |E|)$ time. The second pass also runs in $O(|V| + |E|)$ time, with proof given by

Navarro [78]. Since there are m rows, the runtime of filling the DP matrix is $O(m(|V| + |E|))$.

Algorithm 5 Semi-global Navarro's algorithm

```
Input: a sequence graph $G = (V, E, \sigma)$ and a string $s = \Sigma^m$
Output: DP table $C$ representing the edit distances of the alignment of prefixes of $s$ and paths in $G$
$C \leftarrow (m + 1) \times |V|$-sized matrix of integers initialized with $S_{i,j} = i$

function Propagate($i, u, v$)
    if $C_{i,v} > C_{i,u} + 1$ then
        $C_{i,v} \leftarrow C_{i,u} + 1$
        for $z \in \delta_v^{out}$ do
            Propagate(i, v, z)
        end for
    end if
end function

function Navarro($V, E, \sigma, s$)
    for $i \in [0, .., |s|)$ do
        for $v \in V$ do
            $C_{i+1,v} \leftarrow C_{i,v} + 1$
            for $u \in \delta_v^{in}$ do
                $C_{i+1,v} \leftarrow min(C_{i+1,v}, C_{i,u} + \Delta_{i,v})$
            end for
        end for
        for $v \in V$ do
            for $u \in \delta_v^{out}$ do
                Propagate(i, v, u)
            end for
        end for
    end for
    return $C$
end function
```

Once the DP matrix can be calculated, the score of the optimal alignment is the minimum score in the last row. The optimal alignment can then be extracted by backtracing the DP matrix starting at the minimum score in the last row, similarly to linear alignment.

The runtime can be slightly improved by a pre-processing step [85]. All nodes with an in-degree of 0 and the same label must have the same scores at the end. Therefore these nodes can be merged, producing a graph where $|V| \leq |E| + |\Sigma|$. Since $|\Sigma|$ is in $O(1)$, this ensures that $O(|V|) = O(|E|)$. The preprocessing takes $O(|V| + |E|)$ time and the total runtime is therefore $O(|V| + m|E|)$. This is the optimal runtime, since reading the graph takes $O(|V| + |E|)$ time and filling the DP matrix requires $O(m|E|)$ time unless the strongly exponential time hypothesis is false [79].

The alignment score without the backtrace can be calculated in $O(m + |V| + |E|)$ space since each row only depends on the previous row. The space use of naively implementing Navarro's algorithm with backtrace is $O(m|V| + |E|)$. This can be improved to $O(\sqrt{m}|V| + |E|)$ without an asymptotic penalty to runtime by a technique described previously in linear sequence alignment [16]: calculate the DP table in two passes, first storing each $\sqrt{m}$'th row of the DP table and in the second pass recalculating the rows between. Note that Hirschberg's algorithm [14] cannot be used for graphs since it relies on splitting the DP table into "ahead" and "behind" parts, which is impossible with graphs due to the non-linear structure. Currently it is unknown whether sequences can be aligned to graphs in $O(m + |V| + |E|)$ space and $O(|V| + m|E|)$ time, with no algorithms achieving this limit and no proof that it is impossible.

1.9 Graph indexing

Graph indexing refers to building a data structure over a graph that can then be quickly queried to find exact matches between a query and paths in the graph. Indexing is an important step for sequence-to-graph alignment in practice as the runtime of exact alignment algorithms is infeasible with mammalian sized graphs. While exact substring matching can be done in linear time with strings, exact matching is quadratic in graphs [79], matching the time bounds for approximate matching on graphs. Although asymptotically optimal algorithms for sequence-to-graph alignment require quadratic time [78,79], it is not possible to index a graph in polynomial time such that exact match queries can be achieved in subquadratic time [88]. Indexing graphs is therefore in a sense "more difficult" than aligning to graphs.

Although indexing all paths requires exponential time, indexing methods can still be applied in practice to large graphs by excluding some parts of the graph. Typical approaches are either to remove some nodes or edges, or only supporting queries along specific paths in the graph. Vg toolkit [1], which uses the GCSA index [89], recommends removing high-degree nodes from the graph. Vg also uses a haplotype index [90] when haplotype paths are available, which indexes only sequences contained in the original haplotypes. Pan-Genome Seeding Index [91] finds a set of paths which covers the most k-mers in the graph and only indexes those paths. The missing k-mers can be recovered by indexing the query sequences and traversing the graph, providing polynomial runtime to find all matches in an offline manner. CHOP [92] indexes only sequences present in a set of haplotypes.

Wheeler graphs [93] are a type of graph that can be indexed in polynomial time while supporting linear time queries over all paths in the graph. Wheeler graphs require a total ordering of the nodes such that the graph is *path coherent*: given a contiguous range of nodes $[v_i, ..., v_j]$ and a sequence $s \in \Sigma^n$, the set of nodes reachable from the nodes $[v_i, ..., v_j]$ by the sequence s must also form a contiguous range. A Wheeler graph is essentially a generalization of the Burrows-Wheeler transform to graphs, and any linear string can be considered a linear Wheeler graph. However, Wheeler graphs are a strict subset of directed graphs, and it is NP-hard to detect whether a given directed graph is a Wheeler graph or not [93], and there are deterministic automata where a Wheeler graph with the equivalent language is exponentially larger [94].

1.10 Contribution

In this work I describe my discoveries in sequence-to-graph alignment that enable long reads to be quickly aligned to mammalian scale graphs. First, in Chapter 2 I describe the theoretical basis for rapid sequence-to-graph alignment from my publication “Bit-Parallel Sequence-to-Graph Alignment” [95]. The alignment algorithm described there generalizes Myers’ bit-parallel algorithm to arbitrary graphs. The algorithm led to speedups between 3x-21x over previous sequence-to-graph alignment algorihtms depending on graph topology. With the bit-parallel sequence-to-graph alignment algorithm, optimal alignment to bacterial scale graphs becomes tractable.

Next, in Chapter 3 I describe GraphAligner, a practical tool for aligning long reads to sequence graphs from my publication “GraphAligner: Rapid and Versatile Sequence-to-Graph Alignment” [96]. GraphAligner is based on the bit-parallel sequence-to-graph alignment algorithm presented in Chapter 2. GraphAligner includes theoretical work by generalizing banded alignment to graphs with a dynamic score-based method. The greater part of GraphAligner is however on the practical engineering required to enable long read alignment to scale to mammalian scaled graphs. GraphAligner outperforms existing tools for aligning to sequence graphs in alignment accuracy (96.6% vs 93.8%), and in runtime by a factor of over 13. In addition, I present simple pipelines for genotyping and error correction using GraphAligner. The pipelines use ideas from existing tools but replace graph alignment steps with GraphAligner. The error correction pipeline outperforms existing tools by a factor of 12 in runtime and over 2 in accuracy.

Finally, in Chapter 4 I present two pipelines which use GraphAligner as an integral part. First, AERON is a tool for quantifying RNA expression and detecting fusion genes, published in a joint work with Dilip Durai in “AERON: Transcript quantification and gene-fusion detection using long reads” [97]. AERON aligns long reads to splice graphs with GraphAligner, and recovered known events in the K562 cancer cell line. Second, an unpublished hybrid genome assembly pipeline combining short reads and long reads. The pipeline uses short reads to build a de Bruijn graph and aligns long reads to the graph with GraphAligner, and then uses the alignments to induce overlaps between the long reads.

CHAPTER 2

Bit-parallel matching on graphs

The material in this chapter is re-used from my previous published work "Bit-Parallel Sequence-to-Graph Alignment" [95].

2.1 Bit-parallel exact matching on graphs

The Shift-And algorithm [20–23] is an algorithm for finding exact matches between a query string and substrings of a reference string. This section describes a way to generalize the algorithm to graphs. The generalization proceeds in the same manner as used later in bit-parallel sequence-to-graph alignment and illustrates the principles in a simpler setting. The general idea is to use a DP table, similarly to the linear case, and then fill it in a way that guarantees the runtime does not grow too expensive.

The input is an alphabet Σ, query sequence $s = \Sigma^m$, and a reference graph $G = (V, E \subseteq (V \times V), \sigma : V \to \Sigma)$, where V is the node set, E is a set of directed edges and σ is a function that appoints one character to each node as the node label. The output is a DP table representing the exact matches between all prefixes of the query string and paths in the graph. The DP table directly represents the end positions of the exact matches, and the matching paths can be recovered from the DP table by backtracing.

The generalization requires two insights: first, how to handle nodes with an in-degree greater than one, and second, how to handle cycles. Handling these two cases is sufficient for handling *any* directed graphs.

2.1.1 DAGs

In directed acyclic graphs (DAGs), the nodes are ordered topologically and then processed in topological order. The algorithm for DAGs illustrates the principle of how to handle nodes with an in-degree of greater than one without the complication of cycles. If a node has an in-degree of 1, then the update proceeds in the same way as in the classical Shift-And algorithm: The previous bitvector state (i.e. the state after processing the in-neighbor) is updated according to the label of the present node. However, some nodes have an in-degree of more than 1. For handling such nodes, the bitvector state must first be propagated from each in-neighbor separately. That is, the updated state is computed as if this node was the only in-neighbor. Then the resulting states need to be *merged* such that any exact match from any in-neighbor translates to a match in the node. Here, the invariant to be maintained is that bit i in the bitvector is set after processing a given node if and only if there is a path of length i ending in this node and matching a length-i prefix of the pattern. Since the matching path can come from any of the in-neighbors, and a valid path from any of the in-neighbors translates to a valid path in the node, this invariant can be accommodated by merging the "incoming states" using a bitwise OR operation. Since the merging is a $O(\lceil \frac{m}{w} \rceil)$-time operation, the overall time complexity is unchanged.

2.1.2 Cycles

Handling cycles requires an additional insight from DAGs. Since there is no topological ordering for cyclic graphs, the nodes are instead processed in a non-topological order that allows a node to be processed multiple times. The state of each node is kept throughout the algorithm, and can be updated multiple times until no more changes are necessary. However, by keeping a list of nodes which need to be updated, the runtime can still be bounded. Remember that the invariant in the DP table is that a set bit corresponds to an exact match, and an unset bit means there is no match. In the algorithm for cyclic graphs, this invariant is relaxed during computation: a set bit still corresponds to an exact match, but an unset bit may or may not have a match. At the end of computation, the invariant once again corresponds to the linear case, and an unset bit means there is no match.

Algorithm 6 shows the pseudocode for Shift-And for graphs. The algorithm requires a data structure that allows inserting elements in constant time, and removing an element in constant time. The order of removal specifically does not matter. The nodes are kept in a *calculable queue*, describing nodes which need to be processed. Initially all nodes are in the calculable queue. The invariant of the calculable queue is that any node which might propagate a match forward is contained in the queue. Nodes in the queue must not necessarily be able to propagate a match, but nodes which are not in the queue definitely cannot propagate a match. Nodes are removed from the calculable queue one at a time and processed (line 10). When a node is removed, its state is propagated to all out-neighbors and merged with the existing state. If the state of an out-neighbor changes, it is added to the calculable queue (line 14). Once the calculable queue is empty, no more matches can be extended and the DP table has converged to the correct values.

THEOREM 1 (Completeness of Algorithm 6) At the end of Algorithm 6, if there is an exact match ending at the j'th character in the query sequence s and node i in the reference graph, then the bit $S[i]_j$ is set.

Proof: Assume that there is a match between a $(j+1)$-length prefix of s and a path $(v_0, v_1, .., v_j)$ in the reference graph, where the i'th character of s matches to node v_i in the graph.

Assume that the bit $S[v_k]_k$ is unset for some $k \leq j$. Consider two possible cases: either $k = 0$

Algorithm 6 Shift-And for cyclic graphs

```
1: Input: a sequence graph $(V, E, \sigma)$ and a string $s$
2: Output: DP table $S$ containing the bitvector states of $V$
3: $P \leftarrow |\Sigma|$-sized array of integers initialized with $0$
4: for $i \in [0, .., |s|)$ do                    ▷ Precompute the pattern bitvectors
5:     $P[s_i] \leftarrow P[s_i]$ OR $(1 \ll i)$
6: end for
7: $L \leftarrow$ a list initialized with V
8: $S \leftarrow |V|$-sized array of integers initialized with $S[v] = 1$ if $\sigma(v) = s_0$ and $0$ otherwise
9: while $|L| > 0$ do
10:     $v \leftarrow L.pop()$
11:     for $y \in \delta_v^{\text{out}}$ do
12:         $old \leftarrow S[y]$
13:         $S[y] \leftarrow S[y]$ OR $(((S[v] \ll 1) + 1)$ AND $P[\sigma(y)])$
14:         if $S[y] \neq old$ then
15:             $L.push(y)$
16:         end if
17:     end for
18: end while
```

or $k > 0$. For $k = 0$, the first character of the query string matches one node of the reference graph. Since Line 8 initializes the bit $S[v_0]_0$ based on whether the first character matches the node or not, the bit must have been set. Therefore there is a contradiction and the bit $S[v_k]_k$ cannot be unset when $k = 0$.

For $k > 0$, since the bit $S[v_0]_0$ is set, there must be an index $0 < i \leq k$ such that $S[v_i]_i$ is unset but $S[v_{i-1}]_{i-1}$ is set. Since the bit $S[v_{i-1}]_{i-1}$ is set, the node v_{i-1} must have changed its state at some point, which would add node v_{i-1} to the calculable queue. Once the node v_{i-1} was then popped from the calculable queue, it must have propagated the match to the node v_i and set the bit $S[v_i]_i$. Therefore there is a contradiction and the bit $S[v_k]_k$ cannot be unset when $j > 0$.

From this it follows that the bit $S[v_k]_k$ must be set for all $k \leq j$. ■

THEOREM 2 (Correctness of Algorithm 6) At the end of Algorithm 6, if the bit $S[i]_j$ is set, then there is an exact match ending at the j'th character in the query sequence and node i in the reference graph.

Proof: Assume that the bit $S[i]_j$ is set but there is no path ending at node i which matches the $(j+1)$-length prefix of s. Due to the initialization in Line 8, the first row is correct and only the case $j > 0$ needs to be considered.

By following the backtrace starting at the corresponding node and query character (i, j), we will run into one of two cases: either a bit $S[i']_{j'}$, $0 < j' \leq j$, was set when all bits $S[i'']_{j'-1}, i'' \in \delta_{i'}^{in}$ are unset, that is, a bit was set without a corresponding set in-neighbor bit, or a bit $S[i']_{j'}$, which represents a false positive match, was set from a bit $S[i'']_{j'-1}$ where $S[i'']_{j'-1}$ represents a correct match. The first case, a bit being set without a corresponding set in-neighbor bit, cannot happen as the algorithm only propagates matches forward. In the second case, there is a j'-length match ending at i'' but no $(j'+1)$-length match ending at i'. The only way for this to happen is that the $(j'+1)$'th character of s does not match i'. In that case, Line 13 will not set the bit because the j''th bit of the pattern bitvector $P[\sigma(i')]$ is unset. There is therefore a contradiction and there must be a match ending at the node i' and the j''th character of s. ■

Theorems 1 and 2 prove the correctness of Algorithm 6. To analyze the runtime of Algorithm 6, we note that since the state of a node can only change from a non-match to a match, a node may be updated at most m times. The runtime of Algorithm 6 is therefore also bounded. Updating a node x takes $O(\delta_x^{out})$ time, and each node may be updated at most m times. The total runtime is therefore $O(|V| + m\Sigma_{x\in V}|\delta_x^{\text{out}}|) = O(|V| + m|E|)$. This corresponds to building the DP table cell-by-cell, so in the worst case Algorithm 6 degenerates into a cell-by-cell algorithm but it never grows slower than cell-by-cell updates. Note that for DAGs, each node is only updated once, meaning that the runtime is always faster than cell-by-cell computation by a factor of w.

2.2 Bit-parallel sequence-to-graph alignment

Myers' bitvector algorithm [9] is an optimization of the Needleman-Wunsch algorithm for sequence alignment. This section describes how to generalize Myers' algorithm for graphs. The generalization follows a similar method as the generalization of Shift-And, and requires handling the same two cases: nodes with an in-degree greater than one, and cycles.

2.2.1 DAGs

Similarly to Shift-And for graphs, bit-parallel sequence-to-graph alignment starts by ordering the nodes topologically. Then, the states of the nodes are propagated in a topological order. The difference is that instead of considering only exact matches, all alignments are considered and the values of the DP table represent the edit distance of the best alignment that ends at the specific query character and reference graph node. The merge operation is also different.

The nodes are processed in a topological order. Whenever a node has multiple in-neighbors, the values are first propagated from each in-neighbor separately. Then, the incoming values are *merged* by taking the minimum value among the incoming columns for each cell.

2.2.2 Column merge

Figure 2.1 illustrates the input and output of the merge operation. Given two input columns A and B both with w cells, an output column O is produced such that the score at each cell is the minimum of the two input columns, that is, $S_i^O = min(S_i^A, S_i^B)$ for $i \in [0, .., w)$. There are two algorithms for this, one asymptotically faster with a runtime of $O(\log w)$, and an another with a runtime of $O(w)$ but which runs faster in practice.

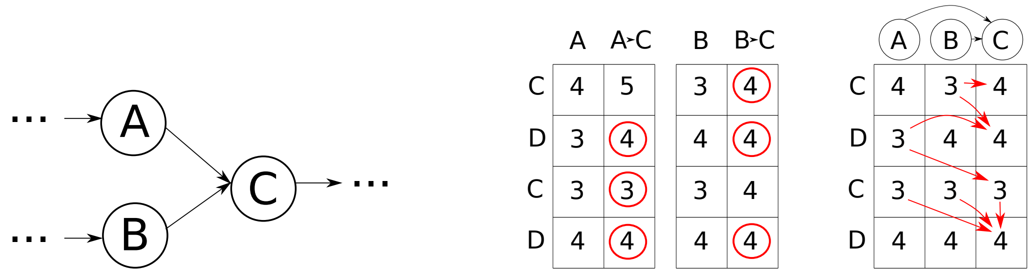


FIGURE 2.1 Handling nodes with an in-degree higher than one in the bitvector framework. Left: The node C has two in-neighbors, A and B. Middle: Each in-neighbor column is separately calculated to get the scores of Recurrence (1.4). The circled cells are the minimum of each row. Right: The resulting column are merged, taking the minimum of the two scores for each row. The arrows show the possible backtraces for each cell.

The first step to merging two columns is to calculate two *difference masks* $M_{A>B}$ and $M_{B>A}$. The masks represent cells where the score of A is larger than B, and vice versa respectively. The masks are then used to merge the bitvectors in $O(1)$ time. The two column merging algorithms differ in how they calculate the difference masks.

The $O(\log w)$ algorithm splits a machine word into $O(\frac{w}{\log w})$ blocks, which are used to simulate registers for calculating $O(\frac{w}{\log w})$ cells simultaneously. The algorithm essentially uses a w-bit machine word to simulate $O(\frac{w}{\log w})$ SIMD registers, each with $O(\log w)$ bits, using regular arithmetic and bitwise operations. Figure 2.2 shows an overview of the algorithm, and Algorithm 7 shows the pseudocode. At each step each block contains the score difference $S_i^A - S_i^B$ for some i. The blocks are first initialized with the score difference at every $O(\log w)$'th cell in $O(\log \log w)$ time. Then, at each step, the difference masks are updated at every $O(\log w)$'th cell in $O(1)$ time. The blocks are then updated in parallel to move one bit forward, such that a block which represented the score difference $S_i^A - S_i^B$ now represents the score difference $S_{i+1}^A - S_{i+1}^B$ instead, in $O(1)$ time. Each individual step takes $O(1)$ time and there are $O(\log w)$ steps in total, so the asymptotic runtime is $O(\log w)$. The output of this are the two difference masks $M_{A>B}$ and $M_{B>A}$.

Figure 2.2 shows the steps of the $O(\log w)$ difference mask algorithm. The input are two bitvectors A and B, consisting of values VP^A, VN^A, S_{end}^A, $S_{before}^A = S_{end}^A + popcount(VN^A) - popcount(VP^A)$ and VP^B, VN^B, S_{end}^B, $S_{before}^B = S_{end}^B + popcount(VN^B) - popcount(VP^B)$ (step A). We assume without loss of generality that $S_{before}^A \leq S_{before}^B$. These bitvectors implicitly represent the values S_i^A and S_i^B (step B). The output is the bitvector representation (VP^O, VN^O, S_{end}^O) of a column S_O such that its values are the minimum of the two columns represented by the input bitvectors, that is, $\forall i \in \{0, 1, ..., w-1\} : S_i^O = \min(S_i^A, S_i^B)$ (step C).

First, we need to find *difference masks* $M_{A>B}$ and $M_{B>A}$, which describe cells where the score of A is higher than B and vice versa (step D). To do this, we first verify that the score differences $S^A - S^B$ are in the range $(-2w, 2w)$ (lines 23-31). We need the $popcount$ operation for this; in most processors an $O(1)$ specialized instruction exists, otherwise it can be calculated in $O(\log w)$ with

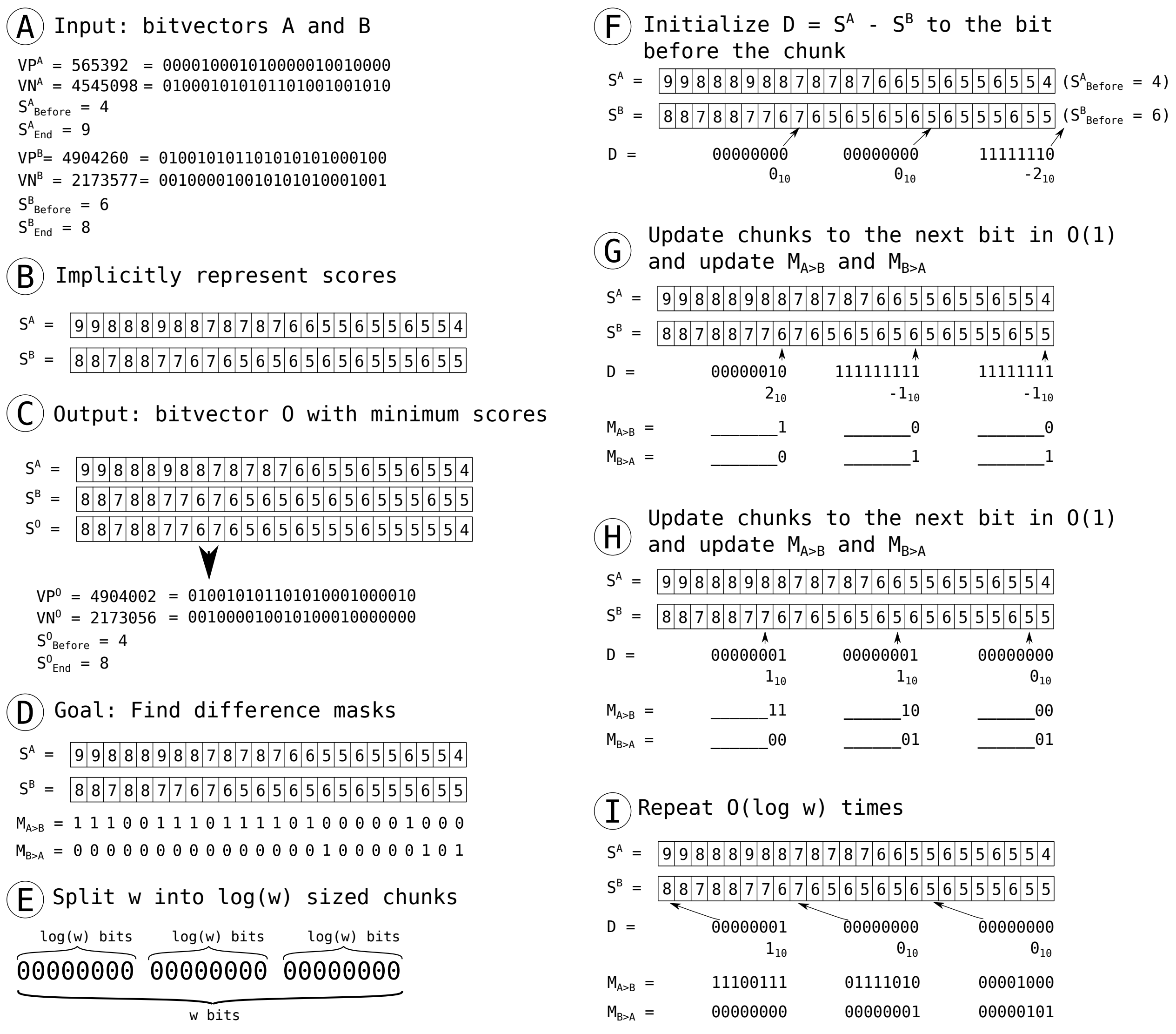

A Input: bitvectors A and B
VP^A = 565392 = 000010001010000010010000
VN^A = 4545098 = 010001010101101001001010
S^A_Before = 4
S^A_End = 9
VP^B= 4904260 = 010010101101010101000100
VN^B = 2173577= 001000010010101010001001
S^B_Before = 6
S^B_End = 8
B Implicitly represent scores
S^A = 9 9 8 8 8 9 8 8 7 8 7 8 7 8 7 6 6 5 5 6 5 5 6 5 5 4
S^B = 8 8 7 8 8 7 7 6 7 6 7 6 5 6 5 6 5 6 5 6 5 5 5 6 5 5
C Output: bitvector O with minimum scores
S^O = 8 8 7 8 8 7 7 6 7 6 7 6 5 6 5 6 5 5 5 6 5 5 5 5 5 4
VP^O = 4904002 = 010010101101010001000010
VN^O = 2173056 = 001000010010100010000000
S^O_Before = 4
S^O_End = 8
D Goal: Find difference masks
M_A>B = 1 1 1 0 0 1 1 1 0 1 1 1 1 0 1 0 0 0 0 0 1 0 0 0
M_B>A = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1
E Split w into log(w) sized chunks
log(w) bits
00000000 00000000 00000000
w bits
F Initialize D = S^A - S^B to the bit before the chunk
(S^A_Before = 4)
(S^B_Before = 6)
D = 00000000 0_10 00000000 0_10 11111110 -2_10
G Update chunks to the next bit in O(1) and update M_A>B and M_B>A
D = 00000010 2_10 111111111 -1_10 11111111 -1_10
M_A>B = _______1 _______0 _______0
M_B>A = _______0 _______1 _______1
H Update chunks to the next bit in O(1) and update M_A>B and M_B>A
D = 00000001 1_10 00000001 1_10 00000000 0_10
M_A>B = ______11 ______10 ______00
M_B>A = ______00 ______01 ______01
I Repeat O(log w) times
D = 00000001 1_10 00000000 0_10 00000000 0_10
M_A>B = 11100111 01111010 00001000
M_B>A = 00000000 00000001 00000101

FIGURE 2.2 Steps of the $O(\log w)$ bitvector merging algorithm. A, B: Two bitvectors are taken as input, which implicitly represent the scores of a column. C: The desired output is a bitvector where the score on each cell is the minimum of the two input bitvectors in that cell. D: Difference masks are used to merge the bitvectors. The difference masks describe which of the two bitvectors has a greater value at the cell. E: A machine word with w bits is used to simulate multiple registers with $O(\log w)$ bits each. F: The variable D is split into chunks, where each chunk represents the score difference $S^A - S^B$ of one cell, evenly spaced $O(\log w)$ cells apart. G, H, I: The chunks are "moved forward" by one bit, such that a chunk which represented the score difference at the i'th cell now represents the score difference at the $i+1$'th cell. After this, the two difference masks are updated at the corresponding cells. This is repeated $O(\log w)$ times to update the difference masks fully.

Algorithm 7 The $O(\log w)$ difference mask algorithm

```
1: $chunkSize \leftarrow (\log_2 w) + 2$ rounded up to the nearest power of two
2: $M_{lsb} \leftarrow$ a constant mask which has 1 at each chunk's least significant bit and 0 elsewhere
3: $M_{sign} \leftarrow$ a constant mask which has 1 at each chunk's sign bit and 0 elsewhere
4: **function** CPFS($x, extra$)    ▷ Chunk prefix sums
5:     $x \leftarrow$ popcount for each $chunkSize$-bit block
6:     $x \leftarrow (x \ll chunkSize) + extra$
7:     $x \leftarrow x * M_{lsb}$    ▷ $x$ now has the prefix sum of the set bits at each chunk boundary
8:     **return** $x$
9: **end function**
10: **function** AddC($x, y$)    ▷ Calculates the values $x + y$ at each chunk in parallel
11:     $signs \leftarrow x \,\&\, M_{sign}$
12:     $x \leftarrow ((x \,\&\, {\sim}M_{sign}) + y) \wedge signs$
13:     **return** $x$
14: **end function**
15: **function** DeductC($x, y$)    ▷ Calculates the values $x - y$ at each chunk in parallel
16:     $signs \leftarrow x \,\&\, M_{sign}$
17:     $x \leftarrow (x|M_{sign}) - y$
18:     $signs \leftarrow signs \wedge (M_{sign} \,\&\, {\sim}x)$
19:     $x \leftarrow (x \,\&\, {\sim}M_{sign})|signs$
20:     **return** $x$
21: **end function**
22: **function** DifferenceMasks($VP^A$, $VN^A$, $S^A_{before}$, $VP^B$, $VN^B$, $S^B_{before}$)
23:     **if** $S^B_{before} - S^A_{before} > popcount(VN^B) + popcount(VP^A)$ **then**
24:         **return** $M_{A>B} = 0^w, M_{B>A} = 1^w$
25:     **end if**
26:     **if** $S^B_{before} - S^A_{before} = 2w$ **and** $VN^B = 1^w$ **and** $VP^A = 1^w$ **then**
27:         **return** $M_{A>B} = 0^w, M_{B>A} = 1^w \,\&\, {\sim}(1 \ll (w-1))$
28:     **end if**
29:     **if** $S^B_{before} - S^A_{before} = 0$ **and** $VN^B = 1^w$ **and** $VP^A = 1^w$ **then**
30:         **return** $M_{A>B} = 1^w, M_{B>A} = 0^w$
31:     **end if**
32:     $D^A \leftarrow (M_{sign} +$ CPFS$(VN^A, 0) -$ CPFS$(VN^B, 0)) \wedge M_{sign}$
33:     $D^B \leftarrow (M_{sign} +$ CPFS$(VN^B, S^B_{before} - S^A_{before}) -$ CPFS$(VN^B, 0)) \wedge M_{sign}$
34:         ▷ $D^A$ and $D^B$ now contain the values $S^A - S^A_{before}$ and $S^B - S^A_{before}$ at each chunk boundary in
    two's complement
35:     $M_{smear} \leftarrow ((D^B \,\&\, M_{sign}) \gg (chunksize - 1)) * ((1 \ll (chunksize - 1)) - 1)$  ▷ 1 where $D^B$
    needs to be deducted from $D^A$ and 0 elsewhere
36:     $D \leftarrow$ AddC(DeductC($D^A, D^B \,\&\, {\sim}M_{smear} \,\&\, M_{sign}$), ${\sim}D^B \,\&\, M_{smear} + M_{smear} \,\&\, M_{lsb}$)   ▷ $D$
    now contains the value difference $S^A - S^B$ at each chunk boundary
37:     **for** $i \in [0, chunkSize - 1]$ **do**    ▷ Calculate the result masks at each $chunkSize$'th bit in parallel
38:         $D \leftarrow$ AddC($D, (VP^A \gg i) \,\&\, M_{lsb} + (VN^B \gg i) \,\&\, M_{lsb}$)
39:         $D \leftarrow$ DeductC($D, (VN^A \gg i) \,\&\, M_{lsb} + (VP^B \gg i) \,\&\, M_{lsb}$)
40:         $M_{D<0} \leftarrow D \,\&\, M_{sign}$
41:         $M_{D\neq 0} \leftarrow ((D|M_{sign}) - M_{lsb}) \,\&\, M_{sign}$
42:         $M_{B>A} \leftarrow M_{B>A}|(M_{D<0} \gg (chunkSize - i - 1))$
43:         $M_{A>B} \leftarrow M_{A>B}|(M_{D\neq 0} \,\&\, {\sim}M_{D<0} \gg (chunkSize - i - 1))$
44:     **end for**
45:     **return** $M_{A>B}, M_{B>A}$
46: **end function**
```

normal arithmetic operations. We define a variable D split in *chunks*, where each chunk represents the score difference $S^A - S^B$ at a certain index. Each chunk is represented by $\log_2 w + 2$ bits and stores a value in two's complement. In this way, each chunk can accommodate a score difference, which takes a value between $-2w$ and $2w$. The chunks essentially simulate a SIMD-like architecture, with the chunks corresponding to SIMD registers, and parallel addition, substraction and comparison to zero instructions implemented with normal arithmetic and bitwise operations. This enables calculating the score difference $S^A - S^B$ in parallel. The chunks of D are first initialized to point at the bit before the start of the chunk (step F, lines 32-36).

Then D is updated in parallel to point at the next bit (step G, lines 38-39). The bitvectors VP^A, VP^B, VN^A and VN^B represent the score difference between two cells, and are used to update D. Since D represents the current score difference $S^A - S^B$, we can update it by $D' = S^A_{i+1} - S^B_{i+1} = S^A_i - S^B_i + (S^A_{i+1} - S^A_i) - (S^B_{i+1} - S^B_i) = D + (VP^A_{i+1} - VN^A_{i+1}) - (VP^B_{i+1} - VN^B_{i+1})$. This is implemented by shifting the appropriate bits from the VP and VN bitvectors to the start of the chunk and selecting the least significant bit, essentially initializing the chunk-registers as either 0 or 1, which are then added and substracted to D (step G, lines 38-39).

Once D has been updated to the next bit, $M_{A>B}$ and $M_{B>A}$ can be updated based on the values in the chunks of D (step G, lines 40-43). Since the chunks of D represent the score difference, $M_{A>B}$ must be set at indices where the value of D is greater than 0, and $M_{B>A}$ where the value is less than 0. As the values are stored in two's complement, the case $D < 0$ is checked by selecting the sign bit and shifting it to the current position (lines 40 and 42). To check $D > 0$, we select indices where D is not negative and not zero (lines 41 and 43).

Updating D, $M_{A>B}$ and $M_{B>A}$ is then repeated $O(\log w)$ times to solve the full masks $M_{A>B}$ and $M_{B>A}$ (step I, lines 37-44). Initializing the variable D takes $O(\log \log w)$ time due to the chunk popcounts in line 5. Each iteration of the parallel update takes $O(1)$ time, and there are $O(\log w)$ iterations. The runtime of calculating the difference masks is therefore $O(\log w)$.

The $O(w)$ algorithm instead first calculates masks that represent how the score difference $S^A_i - S^B_i$ changes. Algorithm 8 shows the pseudocode for this. Given the score difference change $c = (S^A_i - S^B_i) - (S^A_{i-1} - S^B_{i-1})$, there are four masks, two corresponding to the case that $c \in \{-2, 2\}$ (Lines 8 and 11), one mask for the case $c > 0$ (Line 9), and one for $c < 0$ (Line 10). Then, the masks can be used to update the difference masks for a contiguous block. Assuming that the scores at the start are equal, that is, $S^A_{before} = S^B_{before}$, and given the first indices i where the score difference change is positive ($c > 0$) and j where it is negative ($c < 0$), if $i < j$ then $M_{A>B}$ must be set at each index $i \leq x < j$, and similarly for $i > j$ and $M_{B>A}$. Since the score difference $S^A_{before} - S^B_{before}$ is 0, and grows at index i, it must be positive starting from index i, and as the first index where it shrinks is j, S^A_x must be larger than S^B_x in the indices in between. Note that S^A_j might still be larger than S^B_j at index j. The bitvectors are first pre-processed to handle cases where $S^A_{before} \neq S^B_{before}$ in Line 13. Note that a similar block is required for the case $S^B_{before} < S^A_{before}$, which has the labels A and B switched around but is otherwise identical and is not shown to avoid cluttering the pseudocode. Then, in the loop starting at Line 27, bits in the four masks are “neutralized” starting from the least significant index, and each neutralization updates either $M_{A>B}$ or $M_{B>A}$. When neutralizing a bit, if the least significant bits of the $c = 2$ and $c > 0$ masks are at the same index, the bit in the $c = 2$ mask is unset, and otherwise the bit in the $c > 0$ mask is unset. One bit from $c = -2$ or $c < 0$ is similarly unset. The loop runs until the $c > 0$ or $c < 0$ mask is empty. Since the four masks can have up to $2w$ set bits in total, the longest runtime is when each mask has $\frac{w}{2}$ set bits, and the loop can run at most w times. The asymptotic runtime is therefore $O(w)$.

Once we have the difference masks $M_{A>B}$ and $M_{B>A}$, we can use them to merge the bitvectors. Algorithm 9 shows the pseudocode for this. Given an index i, if $S_i^A \geq S_i^B$ and $S_{i-1}^A \geq S_{i-1}^B$, then the output bitvectors can be selected as $VP_i^O = VP_i^A$ and $VN_i^O = VN_i^A$, and vice versa if $S_i^B \geq S_i^A$ and $S_{i-1}^B \geq S_{i-1}^A$. However we need to handle the case where $S_i^A > S_i^B$ and $S_{i-1}^B > S_{i-1}^A$ or symmetrically $S_i^B > S_i^A$ and $S_{i-1}^A > S_{i-1}^B$. In this case, $S_i^A = S_{i-1}^B$, $VN_i^A = 1$, $VP_i^B = 1$ and $VP_i^O = VN_i^O = 0$. We first calculate a picking mask M_p which determines whether a bit needs to be taken from S^A (line 3). The picking mask is set to 1 whenever $M_{A>B} = 1$ and 0 to whenever $M_{B>A} = 1$, and in other indices copies the neighboring bit from the least significant direction. Then we pick the values for VP^O and VN^O based on the picking mask. To handle the special case where $S_i^A > S_i^B$ and $S_{i-1}^B > S_{i-1}^A$, we reduce the VN^A and VN^B vectors such that in those indices the output bitvector’s value cannot decrease (lines 4-5). Merging bitvectors, given the difference masks, is $O(1)$. The total runtime is therefore dominated by the difference mask algorithm.

In practice, the $O(\log w)$ difference mask algorithm takes a constant 487 operations, while the $O(w)$ algorithm takes on average 58 operations for $w = 64$. The $O(w)$ algorithm is used in all later experiments.

2.2.3 Changed minimum value

The algorithm for cyclic graphs requires the *changed minimum value* operation. Given an *old bitvector* A and *new bitvector* B, the changed minimum value is the minimum value among the cells where the value in the new bitvector is smaller, that is, changedMin $= min_{i, S_{i,A} < S_{i,B}}(S_{i,A})$. Figure 2.3 shows the minimum changed value. When none of the cells have a strictly smaller value, the changed minimum value is infinite. Similarly to the column merge operation, there are two algorithms for calculating the changed minimum value, one asymptotically faster with a runtime of $O(\log w)$ and another with a runtime of $O(w)$ that is faster in practice.

The $O(\log w)$ algorithm uses a similar approach as the $O(\log w)$ algorithm for merging bitvectors. A w-bit machine word is split into $O(\frac{w}{\log w})$ chunks, each with $O(\log w)$ bits. The algorithm keeps three words, one for the score difference $S_{i,A} - S_{i,B}$, calculated exactly the same as the word in the bitvector merge algorithm, a second word for the score $S_{i,A} - S_{0,A}$, and a third for a minimum score among $S_{i,A} - S_{0,A}, i \in [k \log w, .., (k+1) \log w) \wedge S_{i,A} < S_{i,B}$. Whenever $S_{i,A} - S_{i,B}$ is negative, the score from the second word is used to update a new minimum for the third word. At the end, there are $O(\frac{w}{\log w})$ minimum scores. The minimum scores can be merged recursively in

Algorithm 8 The $O(w)$ difference mask algorithm

```
1:  function DifferenceMasks($VP^A$, $VN^A$, $S^A_{before}, S^A_{end}$, $VP^B$, $VN^B$, $S^B_{before}, S^B_{end}$)
2:      $c \leftarrow \sim(VP^A \ \& \ VP^B)$
3:      $VP^A \leftarrow VP^A \ \& \ c$
4:      $VP^B \leftarrow VP^B \ \& \ c$
5:      $c \leftarrow \sim(VN^A \ \& \ VN^B)$
6:      $VN^A \leftarrow VN^A \ \& \ c$
7:      $VN^B \leftarrow VN^B \ \& \ c$
8:      $twosmaller \leftarrow VN^A \ \& \ VP^B$
9:      $smaller \leftarrow (VP^B \ \& \sim VN^A) \mid (VN^A \ \& \sim VP^B) \mid (VN^A \ \& \ VP^B)$
10:     $bigger \leftarrow (VP^A \ \& \sim VN^B) \mid (VN^B \ \& \sim VP^A) \mid (VN^B \ \& \ VP^A)$
11:     $twobigger \leftarrow VN^B \ \& \ VP^A$
12:     $M_{A>B} \leftarrow M_{B>A} \leftarrow 0$
13:     if $S^A_{before} < S^B_{before}$ then        ▷ Assume without loss of generality that $S^A_{before} \leq S^B_{before}$
14:         for $i \in [1, .., S^B_{before} - S^A_{before})$ do
15:             $lsb \leftarrow bigger \ \& \sim(bigger - 1)$
16:             $bigger \leftarrow bigger \wedge (\sim twobigger \ \& \ lsb)$
17:             $twobigger \leftarrow twobigger \ \& \sim lsb$
18:             if $bigger = 0$ then
19:                 return $M_{A>B} = 0^w$, $M_{B>A} = 1^w$
20:             end if
21:         end for
22:         $lsb \leftarrow bigger \ \& \sim(bigger - 1)$
23:         $M_{B>A} \leftarrow M_{B>A} \mid (lsb - 1)$
24:         $bigger \leftarrow bigger \wedge (\sim twobigger \ \& \ lsb)$
25:         $twobigger \leftarrow twobigger \ \& \sim lsb$
26:     end if
27:     for $i \in [0, .., w]$ do
28:         if $smaller = 0$ or $bigger = 0$ then
29:             if $smaller = 0$ then
30:                 $lsb \leftarrow bigger \ \& \sim(bigger - 1)$
31:                 $M_{A>B} \leftarrow M_{A>B} \mid -lsb$
32:             else if $bigger = 0$ then
33:                 $lsb \leftarrow smaller \ \& \sim(smaller - 1)$
34:                 $M_{B>A} \leftarrow M_{B>A} \mid -lsb$
35:             end if
36:             break
37:         end if
38:         $lsbbigger \leftarrow bigger \ \& \sim(bigger - 1)$
39:         $lsbsmaller \leftarrow smaller \ \& \sim(smaller - 1)$
40:         if $lsbbigger > lsbsmaller$ then
41:             $M_{B>A} \leftarrow lsbbigger - lsbsmaller$
42:         else
43:             $M_{A>B} \leftarrow lsbsmaller - lsbbigger$
44:         end if
45:         $bigger \leftarrow bigger \wedge (\sim twobigger \ \& \ lsbbigger)$
46:         $twobigger \leftarrow twobigger \ \& \sim lsbbigger$
47:         $smaller \leftarrow smaller \wedge (\sim twosmaller \ \& \ lsbsmaller)$
48:         $twosmaller \leftarrow twosmaller \ \& \sim lsbsmaller$
49:     end for
50:     return $M_{A>B}$, $M_{B>A}$
51: end function
```

Algorithm 9 The bitvector merge algorithm

```
1: function MergeBitvectors($VP^A$, $VN^A$, $S^A_{before}$, $S^A_{end}$, $VP^B$, $VN^B$, $S^B_{before}$, $S^B_{end}$)
2:     $M_{A>B}, M_{B>A} \leftarrow$ DifferenceMasks($VP^A$, $VN^A$, $S^A_{before}$, $VP^B$, $VN^B$, $S^B_{before}$)
3:     $M_p \leftarrow (M_{A>B} | ((M_{B>A} | M_{A>B}) - (M_{A>B} \ll 1)))$ & $\sim M_{B>A}$
4:     $VN^A_{reduced} \leftarrow VN^A$ & $\sim(M_{B>A}$ & $(M_{A<B} \ll 1))$
5:     $VN^B_{reduced} \leftarrow VN^B$ & $\sim(M_{A>B}$ & $(M_{B<A} \ll 1))$
6:     $VP^{out} \leftarrow VP^B$ & $M_p + VP^A$ & $\sim M_p$
7:     $VN^{out} \leftarrow VN^B_{reduced}$ & $M_p + VN^A_{reduced}$ & $\sim M_p$
8:     return $S^{out}_{end} = \min(S^A_{end}, S^B_{end})$, $VP^{out}$, $VN^{out}$
9: end function
```

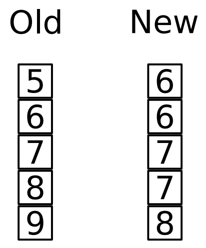


FIGURE 2.3 Changed minimum value. The changed minimum value of the new column is 7, from the fourth row. The topmost row changed, but it is not smaller in the new column so it does not count for the changed minimum value. Similarly the second and third rows do not count since the score did not change.

$O(\log(\frac{w}{\log w})) = O(\log w)$ operations.

The $O(w)$ algorithm uses a similar approach as the $O(w)$ algorithm for merging bitvectors. Algorithm 10 shows the pseudocode. The changed minimum value must be at a location where $S^B_i > S^A_i$, that is, $M_{B>A}$ is set, and the score of A is at a local minimum. First, the difference mask $M_{B>A}$ is calculated using the bitvector merge algorithm. Then, a mask of local minima of A are found by using the bitvectors VP^A and VN^A. Finally, the difference mask and the local minima mask are bitwise-AND-ed to produce a list of indices which must contain the changed minimum value. The value of each of the indices is checked using the equation $S_{i,A} = S_{end,A} + popcount(VN^A_{i+1..w}) - popcount(VP^A_{i+1..w})$ and the minimum is extracted.

The $O(\log w)$ algorithm was never implemented since it works similarly to the $O(\log w)$ bitvector merging algorithm, which is much slower than the $O(w)$ bitvector merging algorithm. In practice, the $O(w)$ minimum changed value algorithm uses on average 142 operations, including calculating the difference mask, and the $O(\log w)$ algorithm would require at least 487 since it uses the $O(\log w)$ bitvector merging algorithm.

2.2.4 Cycles

The method for handling cycles is similar to Shift-And. The algorithm keeps a list of nodes to be calculated, where nodes may be entered multiple times. Then, the states of the nodes are updated until the values have converged to correctness.

Algorithm 10 The $O(w)$ minimum changed value algorithm

```
1: function MinChanged($VP^{new}$, $VN^{new}$, $S^{new}_{before}$, $S^{new}_{end}$, $VP^{old}$, $VN^{old}$, $S^{old}_{before}$, $S^{old}_{end}$)
2:     $M_{new>old}, M_{old>new} \leftarrow$ DifferenceMasks($VP^{new}$, $VN^{new}$, $S^{new}_{before}$, $VP^{old}$, $VN^{old}$, $S^{old}_{before}$)
3:     $possibleMinima \leftarrow (VP^{new} \,\&\, (VN^{new} - VP^{new})) \gg 1$
4:     $possibleMinima \leftarrow possibleMinima \mid ((1 \ll w) \,\&\, (VN^{new} \mid \sim(VN^{new} - VP^{new})) \,\&\, \sim VP^{new})$
5:     if $M_{B>A} \neq 1^w$ then
6:         $possibleMinima \leftarrow possibleMinima \mid ((\sim M_{B>A}) \gg 1) \mid ((\sim M_{B>A}) \ll 1) \mid 1 \mid (1 \ll w)$
7:         $possibleMinima \leftarrow possibleMinima \,\&\, M_{B>A}$
8:     end if
9:     $result \leftarrow \infty$
10:    if $S^{new}_{before} < S^{old}_{before}$ then
11:        $result \leftarrow S^{new}_{before}$
12:    end if
13:    while $possibleMinima \neq 0$ do
14:        $mask \leftarrow possibleMinima \wedge (possibleMinima - 1)$
15:        $result \leftarrow \min(result, S^{new}_{before} + popcount(VP^{new} \,\&\, mask) - popcount(VN^{new} \,\&\, mask))$
16:        $possibleMinima \leftarrow possibleMinima \,\&\, \sim mask$
17:    end while
18:    return $result$
19: end function
```

Unlike Shift-And, where the nodes can be popped in an arbitrary order, generalizing the bit-parallel algorithm to graphs requires the nodes to be popped in a certain order. To achieve this, we use a priority queue and define the operator $push(p, v)$ to either insert or update element v: if the element v does not exist in the priority queue, it is inserted with priority p; if it exists and has a priority higher than p, the priority is updated to p; if it exists and has a priority equal or lower than p, the operation is ignored. We define the *changed minimum score* between two columns as the minimum value among the cells which are different between the two columns, or infinite if the two columns have equal values in all cells, and use $changedMin$ to refer to this operation.

Algorithm 11 shows the pseudocode for the generalization of the bit-parallel algorithm to graphs. The algorithm keeps a lowest-first priority queue L which contains all columns whose values can propagate forward, and the minimum value which might propagate. Initially, all nodes are inserted into L. The nodes are popped from L one at a time, with lowest priority first. Once a node is calculated, it propagates its state to all of its out-neighbors using the column calculate operation from Myers' algorithm [9] as described in Section 1.2.4. The incoming states are then merged with the existing state using the bit-parallel merge operation as described in Section 2.2.2. After this, the changed minimum value is taken between the old and the new state of the column. The minimum changed value represents the minimum value which might be propagated forward from the node. The new node is then inserted to L with the minimum changed value as the priority, if the minimum changed value is not infinite.

To establish the correctness and runtime of the algorithm, we first define the *present score* of a cell to be the score assigned to the cell at some point during the calculation, as opposed to the *correct score* which is the unique value that satisfy the recurrence in Equation 1.4. We say that a

Algorithm 11 Bitvector alignment algorithm for cyclic graphs

```
1: Input: a sequence graph $(V, E, \sigma)$ and a string $s$
2: Output: Vector $S$ containing the column states of $V$
3: $P \leftarrow$ precomputed pattern bitvectors for $\forall c \in \Sigma$ based on $s$
4: $L \leftarrow$ a priority queue initialized with $(0, v), \forall v \in V$
5: $S \leftarrow |V|$ -sized array of bitvectors initialized with $VP = 1^m, VN = 0^m, S_{\text{end}} = m$
6: while $|L| > 0$ do
7:     $(\_, v) \leftarrow L.pop()$
8:     for $y \in \delta^v_{out}$ do
9:         $old \leftarrow S[y]$
10:        $\triangleright$ $\otimes$: merge operation, $F$: bitvector step from [9]
11:        $S[y] \leftarrow S[y] \otimes F(S[v], P_{\sigma(y)})$
12:        if $changedMin(old, S[y]) \neq \infty$ then
13:            $L.push(changedMin(old, S[y]), y)$
14:        end if
15:    end for
16: end while
```

cell has *converged* when its present score is equal to its correct score.

THEOREM 3 In Algorithm 11, if the minimum priority among all items in L is x, then all cells whose correct scores are $C_{i,j} < x$ have converged.

Proof: We show this by induction. For the initial case, there are no cells whose correct scores are negative, so the statement holds when $x = 0$. Next, we will assume that the minimum priority in L is x and that all cells whose correct scores are $C_{i,j} < x - 1$ have converged, and show that all cells whose correct scores are $C_{i,j} = x - 1$ have converged. Assume that there is a cell whose correct score is $x - 1$. There are four cases for how the cell's correct score is defined: (i) the vertical term, (ii) the horizontal term, (iii) the diagonal term with a mismatch, (iv) the diagonal term with a match.

Case (i). The cell has a vertical neighbor $C_{i,j-1}$ whose correct score is $x - 2$. By assumption cells with correct score $C_{i',j'} < x - 1$ have converged, so the vertical neighbor's present score is $x-2$. The bitvector representation allows a vertical score difference of up to 1, so the cell's present score is at most $x - 1$ and the cell has converged.

Case (ii). The cell has a horizontal neighbor $C_{i',j}$ whose correct score is $x-2$. The neighbor cell has converged by assumption. After the last time the neighbor column was calculated, the neighbor cell had its correct score. Since there is a cell with a present score $x-2$ in the neighboring column, the node i' was added to L with a priority of $x-2$ (or less). Therefore the edge (i', i) was processed at some point earlier in the calculation, and at that point Equation (1.4) was applied to the cell $C_{i,j}$, producing the correct score.

Case (iii). Analogous to Case (ii).

Case (iv). The cell has a diagonal neighbor $C_{i',j'}$ whose correct score is $x - 1$. If the diagonal neighbor has converged, then the node i' will have been added to L with a priority of $x - 1$ (or less), and the argument from case (ii) applies. Next we need to prove that the diagonal neighbor has converged. The diagonal neighbor cell's correct score is again defined by the same cases (i)-(iv). For cases (i)-(iii), the diagonal neighbor has converged. For case (iv), we look at the diagonal neighbor cell's diagonal neighbor cell, and keep traversing by diagonal connections until we reach a cell for whom one of cases (i)-(iii) applies. Since the diagonal neighbors cannot form cycles, this will eventually happen, proving that the entire chain has converged. ■

From Theorem 3 it follows that once the minimum priority among all items in L is $m + 1$, all cells have converged to their correct scores, so the algorithm will eventually reach the correct solution in cyclic areas. Next we will establish an upper bound on the time until convergence.

COROLLARY 1 If all cells whose correct scores are $C_{i,j} < x$ have converged, then all cells whose present scores are $C_{i,j} \leq x$ have converged.

Proof: We assumed that all cells whose correct scores are $C_{i,j} < x$ have converged. Therefore there are no cells whose present score is x but whose correct score is $C_{i,j} < x$. A cell's present score cannot be lower than its correct score since the present scores are initialized at the highest possible value and applying Equation (1.4) cannot lower them under the correct score. Therefore if a cell's present score is x, it must also be its correct score. ■

THEOREM 4 A node cannot be popped from L more than m times.

Proof: If a node v is popped from L with a priority x, it was added to the queue with a priority x at some point. This implies that there is at least one cell $C_{v,j}$ in the column with a present score of x. By Theorem 3 all cells with correct scores below x have converged and consequently $C_{v,j}$ has converged by Corollary 1. Therefore each pop of a node v must be preceded by an update to node v's state that causes at least one cell to converge. Since a cell can converge only once, and a column has m cells, this can happen at most m times per node. ■

From Theorem 4, the outer loop starting in line 6 runs at most $m|V|$ times. Since the inner loop in line 8 is processed $|\delta_v^{out}|$ times per outer loop iteration, the inner loop runs at most $m\Sigma_{v\in V}|\delta_v^{out}| = m|E|$ times. This provides a bound of $m|E|$ inner loop iterations, meaning that in the worst case, the cyclic bitvector algorithm behaves like a cell-by-cell algorithm.

Algorithm 11 uses a priority queue to store the calculable nodes. Since the maximum score a cell can have is m, the priority queue can be implemented as m arrays, one for each priority, plus a $|V|$-sized array for the node's current position in the queue for the $push$ operation. In this case inserting and retrieving n values can be done in $O(|V| + m + n)$ time. Since $|V| \leq n \leq m|V|$, the total runtime of inserting and retrieving all n elements is bounded by $O(m|V|)$.

In summary, the inner loop in Line 8 runs $O(m|E|)$ times. The runtime of the inner loop is dominated by the column merge and changed minimum value operations, which both run in time $O(\log w)$. The total runtime is therefore $O(m|E| \log w)$.

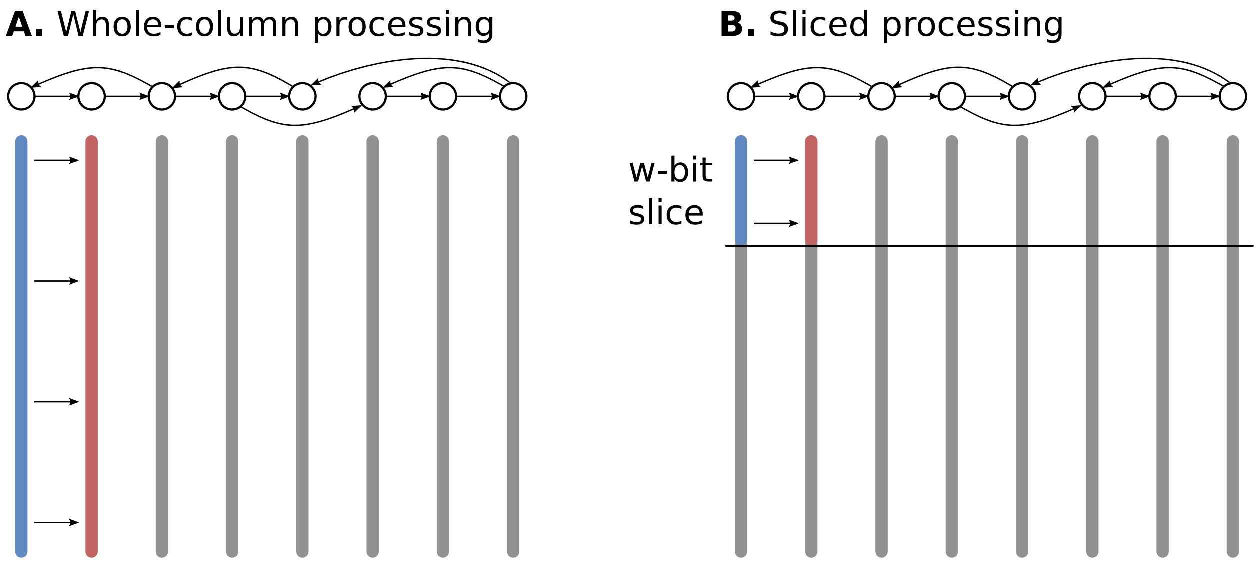


FIGURE 2.4 The DP table for aligning a sequence to a graph (shown on top) is represented by a set of columns (vertical bars), each corresponding to one graph node. The table can be filled in different orders: **A.** each update operation (from blue to red) proceeds on a complete column. **B.** update operations commence on "slices" of w bits; only after the final values in a slice (i.e. for all columns) have been computed, we proceed to the next slice.

2.2.5 Bitvector analysis

The DP table can be filled in two different ways depending on the size of the bitvectors used. Figure 2.4 shows these two ways. In *whole-column processing* (A), the bitvectors are as long as the input sequence. The DP table is calculated one column at a time, where the column spans the whole table. Whole-column processing is analogous to column-wise processing in Myers' algorithm, and for linear graphs, equivalent to it. In *sliced processing* (B), the DP table is partitioned into slices of w rows. Each slice is first calculated until convergence, and after that the next slice is calculated. This is analogous to row-wise processing in Myers' algorithm.

The choice of bitvector length affects the runtime of the algorithm. We classify the operations used in bitwise operations as either *elementary operations*, which are all arithmetic and bitwise operations, and *column operations* which are the column merging and changed minimum value operations. For bitvectors of length k, where w is the size of the machine word, elementary operations are $O(\left\lceil \frac{k}{w} \right\rceil)$ and column operations are $O(\left\lceil \frac{k}{w} \right\rceil \log k)$. Total runtime of the algorithm is therefore $O(m|E| \left\lceil \frac{k}{w} \right\rceil \log k)$ for cyclic graphs and $O(\left\lceil \frac{m}{k} \right\rceil |E| \left\lceil \frac{k}{w} \right\rceil \log k)$ for acyclic graphs. Using $k = w$ reduces these to $O(1)$ for elementary operations and $O(\log w)$ for column operations, and $O(m|E| \log w)$ for cyclic graphs and $O(\left\lceil \frac{m}{w} \right\rceil |E| \log w)$ for acyclic graphs. Smaller k leads to a lower asymptotic runtime for cyclic graphs, all the way until $k = 1$. However, using a small k negates the runtime advantage of using bitvectors and increases the runtime for acyclic graphs. A higher k might also reduce the time to converge in cycles. We chose to use $k = w$ without experimentally measuring the actual runtime for different k due to the difficulty of implementing different

BGSA	Seqan	Our method (whole-column)	Our method (sliced)
1.3s	1.2s	1.5s	5.5s

TABLE 2.1 Sliced versus whole-column processing on a linear graph

versions of the column operations for various values of k. Different values of k can however be easily measured in linear graphs, although this does not use any column operations and therefore might not be representative of the effect on non-linear graphs.

2.2.6 Experiments

To evaluate the bit-parallel alignment algorithm, we performed four experiments. First, we evaluated the effect of sliced versus whole-column processing, and compared the implementation to two standard implementations of Myers' bitvector algorithm for linear alignment. Second, we experimented with different graph topologies to measure their effect on runtime. Third, we evaluated the algorithm with real data, aligning real reads to a pangenome graph of the HLA-A gene. Fourth, we evaluated the algorithm with a larger real dataset, aligning real reads to a de Bruijn graph of *E. coli*.

2.2.6.1 Bitvector performance

The sliced processing (Fig. 2.4) adds extra overhead compared to the whole-column processing used in the classical Myers' algorithm. The reference sequence must be accessed multiple times, and memory use is not cache-efficient, since a large memory range is written and read a few times per address instead of a small range updated many times per address. To measure the overhead added by this, we ran the bitvector algorithm on a graph consisting of a linear chain of nodes with 200 000 bp in total and a 100 000 bp query. This linear graph mimicks sequence-to-sequence alignment and we compared our performance with optimized implementations of Myers' algorithm from BGSA [12] and Seqan [11] on the same sequences. We also tested whole-column processing for the linear graph to see how much of the difference is due to code optimization and how much is due to the different processing methods. Note that BGSA is particularly designed to be fast in the case when multiple reads are aligned in parallel. To facilitate a meaningful comparison, we used BGSA in a mode resembling Myers' bitvector algorithm, i.e. we aligned one read on one CPU without using vector instructions.

Table 2.1 shows the results. The sliced processing method is noticably slower than the optimized implementations or the whole-column method. The whole-column method's performance is close to the optimized implementations, which indicates that our implementation does not incur significant overheads. The overhead of the sliced processing method therefore seems to be inherent to processing non-trivial graphs. In the remaining experiments we use the sliced processing

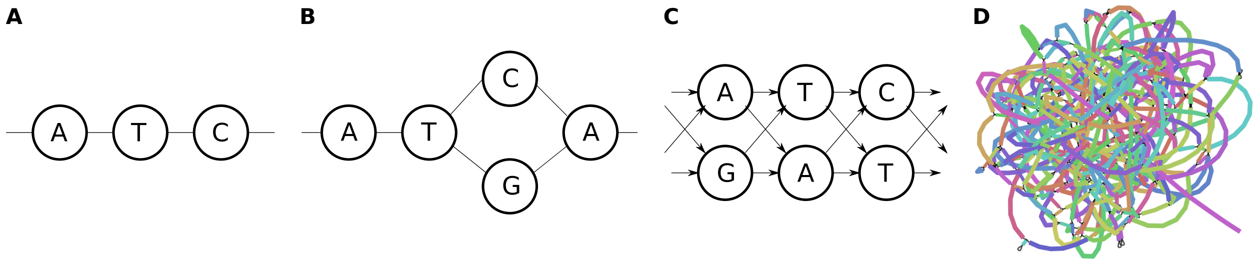


FIGURE 2.5 Overview of the graphs used in the graph topology experiment. A. Linear graph, B. SNP graph, C. twopath graph, D. tangle graph (visualized with Bandage [98]).

method.

2.2.6.2 Graph Topology Experiment

For the graph topology experiment, we created four kinds of graphs (Figure 2.5), representing increasing levels of difficulty, based on the *E. coli* reference genome's 10 000 first base pairs.

The first graph, the *linear graph*, is a linear chain of nodes. Aligning to this graph is equivalent to sequence-to-sequence alignment. The second graph, the *SNP graph*, is a linear chain of nodes with randomly inserted bubbles representing single nucleotide polymorphisms (SNPs). The SNPs are distributed at an average of one SNP per 10 base pairs. The third graph, the *twopath graph*, is an artificial worst case graph for the bitvector algorithm. Each node has two in-neighbors, which means that the bitvector merging algorithm has to run for each node. For the first three graphs, neither algorithm's runtime depends on the matched sequence, so the additional inserted nodes were given random labels. The fourth graph, the *tangle graph*, is based on a de Bruijn graph of the reference sequence with k=11. We chose k to be so small specifically to make the graph very cyclic and tangled.

For the tangle graph, the non-branching areas are merged to unitigs, and overlaps between the nodes are removed by deleting the last $k-1$ characters of each non-tip node, producing a directed node-labeled graph with the same topology and same paths as the original de Bruijn graph. For each graph, we also included the reverse-complement strand to map reads simulated from the backwards strand, doubling the graph size and effectively mimicking a bidirectional graph. The graph sizes in Table 2.2 refer to this doubled bidirectional size.

We simulated reads with 20x coverage (total 200 000 bp) from the reference using PBSIM [99], which produced 65 reads with an average length of 3kbp. In addition, we took a high coverage Illumina dataset[1], filtered them by using minimap2 [29] to select reads which align to the first 10 000bp of the reference, and then randomly sampled a 50.5x coverage subset (5 050 reads, 505 000 bp). Then, we aligned both the simulated long reads and the real short reads to the graphs using both our bitvector algorithm and the cell-by-cell approach.

[1]https://www.ebi.ac.uk/ena/data/view/ERX008638

2.2.6.3 HLA-A Experiment

To assess the algorithm's performance on a more realistic scenario, we built a graph of the human HLA-A gene and aligned real sequencing data to it. We took the 4637 alleles of the human HLA-A gene available from the IMGT/HLA database [100], and computed a multiple sequence alignment between them by using Clustal Omega [101] version 1.2.4 with the command "clustalo -i sequences.fasta –outfmt clustal > aln.clustal". Then we used vg [1] version 1.9.0 to build a variation graph from the multiple sequence alignment with the command "vg construct -M aln.clustal -F clustal -m 32 > msa.vg".

For the sequence data, we used Illumina and PacBio reads from NA19240 [56]. To filter the Illumina reads, we used minimap2 [29] to align the reads to the known alleles, producing 2829 Illumina reads (355 981 bp) with an alignment, which we considered to be from the HLA-A region. For the PacBio reads, we selected those whose alignment to the reference genome overlaps with HLA-A's location, producing 102 reads (405 415 bp). Both the Illumina and PacBio reads were then aligned to the graph using the bitvector and cell-by-cell algorithms.

2.2.6.4 *E. coli* Experiment

For the E. coli experiment, we used sequencing data of *Escherichia coli* strain K-12 substrain MG1655. We took 670x coverage Illumina reads from the European Nucleotide Archive[2] and 144x coverage PacBio reads from the NCBI sequence archive[3]. We built a de Bruijn graph of the Illumina dataset using BCalm [67], with k=31 and k-mer solidity threshold 7. We applied the same postprocessing of the graph as described above for the tangle graph. Then we selected PacBio reads longer than 1000 base pairs and randomly sampled a subset of them corresponding to 1.5$\times$ average genome coverage, and aligned them to the graph with the bitvector and cell-by-cell algorithms.

2.2.6.5 Results

Table 2.2 shows a summary of the results. The first eight rows correspond to the graph topology experiment and the last three to the HLA-A and E. Coli experiments. Each number is an average over 10 runs, showing the total time to align all reads on one CPU core. The bitvector approach is faster than the cell-by-cell approach in each graph. As expected from the time complexity analysis, the difference is greater in the acyclic graphs. For the acyclic graphs, the bitvector algorithm achieves between ten- and twentyfold speed improvement. For the cyclic graph, the speedup is between three- and twelvefold, suggesting that cycles are recalculated on average only a few times (linear speedup divided by cyclic speedup) instead of the theoretical worst case of w times. The HLA-A and E.Coli experiments show that the results generalize to more realistic scenarios as well. Note that in our experiments, we compute the *complete* DP matrix and therefore, the long absolute

[2]https://www.ebi.ac.uk/ena/data/view/ERX008638
[3]https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR1284073

Graph	Reads	Nodes	Edges	Bitvector	Cellwise	Speedup
linear	PBSIM	20 000	19 998	1.2s	23.5s	19.6×
linear	Illumina	20 000	19 998	5.5s	62.5s	11.4×
SNP	PBSIM	22 030	24 058	2.3s	41.8s	18.5×
SNP	Illumina	22 030	24 058	9.0s	106s	11.8×
twopath	PBSIM	40 004	80 000	13.0s	168s	12.9×
twopath	Illumina	40 004	80 000	42.1s	446s	10.6×
tangle	PBSIM	19 814	20 398	8.1s	39.4s	4.8×
tangle	Illumina	19 814	20 398	33.8s	102s	3.0×
HLA-A	PacBio	5 864	9 668	2.4s	51.0s	21.3×
HLA-A	Illumina	5 864	9 668	3.7s	44.5s	12.1×
EColi	PacBio	10 510 252	10 540 270	156 000s	1 860 000s	11.9×

TABLE 2.2 Comparison of the bit-parallel algorithm (Bitvector) with Navarro's algorithm [78] (Cellwise)

time for the *E. coli* experiment are not surprising. In fact, this shows the feasibility of computing *optimal* alignments for bacterial genomes.

2.3 Conclusion

In this chapter I have presented my theoretical work on sequence-to-graph alignment. The bit-parallel DP algorithm described here provides a basis for quickly aligning reads to sequence graphs. However, the DP algorithm by itself is not enough to scale to large, mammalian scale graphs. In the next chapter I will describe how to apply the DP algorithm in practice.

2.4 Acknowledgments

We thank Gonzalo Navarro for fruitful discussions on pattern matching on graphs, in particular on Shift-And extension. We are grateful for Dagstuhl Seminar 16351 on 'Next Generation Sequencing—Algorithms, and Software For Biomedical Applications', which sparked the idea to pursue this topic.

CHAPTER 3

GraphAligner

The material in this chapter is re-used from my previous published work "GraphAligner: Rapid and Versatile Sequence-to-Graph Alignment" [96].

The algorithmic work in the previous section laid the groundwork for efficient sequence-to-graph alignment tools. The algorithm provides a way of calculating the optimal alignment by filling the entire DP matrix. However, this is not enough to scale to whole genome alignment. Heuristic approaches, including banded alignment and seeded alignment, are necessary for this. The following presents the implementation of the algorithm in the tool GraphAligner [96]. GraphAligner includes an important theoretical discovery in extending *banded alignment* [19, 25] to arbitrary graphs. GraphAligner is designed to work with arbitrary sequence graphs, including de Bruijn graphs and variation graphs. It is specialized for long error-prone reads, as tools for aligning long reads to sequence graphs were underdeveloped at the start of the project. GraphAligner uses a seed-and-extend method, using a minimizer [41] based index to find seed hits and the bitvector-based alignment algorithm from the previous section [95] for extension. We perform experiments comparing GraphAligner to state of the art linear and graph aligners and find that GraphAligner is competitive with well-optimized linear aligners and outperforms graph aligners by an order of magnitude in runtime. Finally, to show how improved alignment methods improve downstream pipelines, we present a hybrid error correction pipeline using GraphAligner. The error correction pipeline also outperforms the current state of the art by an order of magnitude in runtime and up to three times in error rates.

3.1 Data formats

GraphAligner uses common data formats for input and output, and specifically aims to be compatible with vg [1] to reuse existing operations and pipelines. Reads are inputed either in fasta [102] or fastq format. The sequence quality field of fastq input is ignored. The input reads can also be optionally compressed with gzip. Graphs are inputed in vg [1] format or the text based gfa [47] format used by many genome assemblers [47, 67, 103, 104]. The gfa format allows nodes with overlapping labels, such as in de Bruijn graphs, which vg currently does not. Alignments can be outputed as vg's gam format [1], an equivalent JSON format, or gaf format used by minigraph [84]. The gam and JSON output contain the same fields encoded as binary or text respectively, while the gaf format has different fields.

3.2 Graph model

GraphAligner inputs *bidirected graphs* [105, 106], which are capable of representing genome graphs commonly used in bioinformatics, including de Bruijn graphs [66,67], assembly graphs [68, 103, 107], pangenomes [60], and variation graphs [1, 65]. Bidirected graphs model the double-stranded nature of DNA. The sequence is stored in the nodes, which can be traversed in two directions; either left to right (*forward*) with the node label, or right to left (*backward*) with the reverse complement of the label. We notate a traversal's *orientation* as $+$ for the forward traversal, and $-$ for the backward traversal. The edges connect to either the *left end* or the *right end* of a node. A path through a bidirected graph enters a node from one end, traverses through the node, and then leaves via an edge in the opposite end. Formally, a bidirected graph can be defined as a $G_b = \{V_b, E_b \subseteq (V_b \times \{+,-\} \times V_b \times \{+,-\} \times \mathbb{N}), \sigma_b : V_b \to \Sigma^n\}$, where V_b is the set of nodes, E_b contains a set of bidirected edges connecting ends of two nodes with an exact overlap, and σ_b is a function assigning a node label to each node in V_b. We define the *opposite* of an orientation as $\bar{+} = -$ and $\bar{-} = +$. A bidirected edge (v_1, o_1, v_2, o_2, n) is equivalent to $(v_2, \bar{o_2}, v_1, \bar{o_1}, n)$ and we define that the set E_b contains both equivalent edges if the input graph contains either of them. We use the notation $\bar{s}$ to mark the reverse complement of a string $s = \Sigma^n$.

The bidirected graph is first converted into a directed node-labeled graph which we call the *alignment graph*. The alignment graph is defined as a directed graph $G_a = (V_a, E_a \subseteq (V_a \times V_a), \sigma_a = V_a \to \Sigma^n)$, where V_a is the set of nodes, E_a is a set of directed edges, and σ_a assigns a node label to each node in V_a.

The bidirected graph allows an exact overlap between edges, representing for example overlapping $k-1$-mers of a de Bruijn graph, or the read overlap in an assembly graph. Here, we consider the edges to be labeled by the number of overlapping nucleotides. When traversing via an edge with an overlap of n nucleotides, the path must skip the first n nucleotides of the target node. The overlaps can also vary between edges. Edge overlaps are handled by chopping the node into pieces at each overlap boundary. The alignment graph then has edges connecting the end of a node to the chopped boundary of the neighbor. This allows a path that ends at one node to enter the neighboring node without traversing the overlap twice. Figure 3.1 shows an example of the edge chopping for edges with variable overlaps.

Formally, given a bidirected node v and a set of incoming left edges $E_+ = \{(u_1, \{+,-\}, v, +, m_1), (u_2, \{+,-\}, v, +, m_2), ...\}$, we define a set of *forward breakpoints* $B_v^+ = \{0, m_1, m_2, ..., |\sigma_b(v)|\}$, and given the set of incoming right edges $E_- = \{(u_1, \{+,-\}, v, -, m_1), (u_2, \{+,-\}, v, -, m_2), ...\}$ define a set of *backward break-*

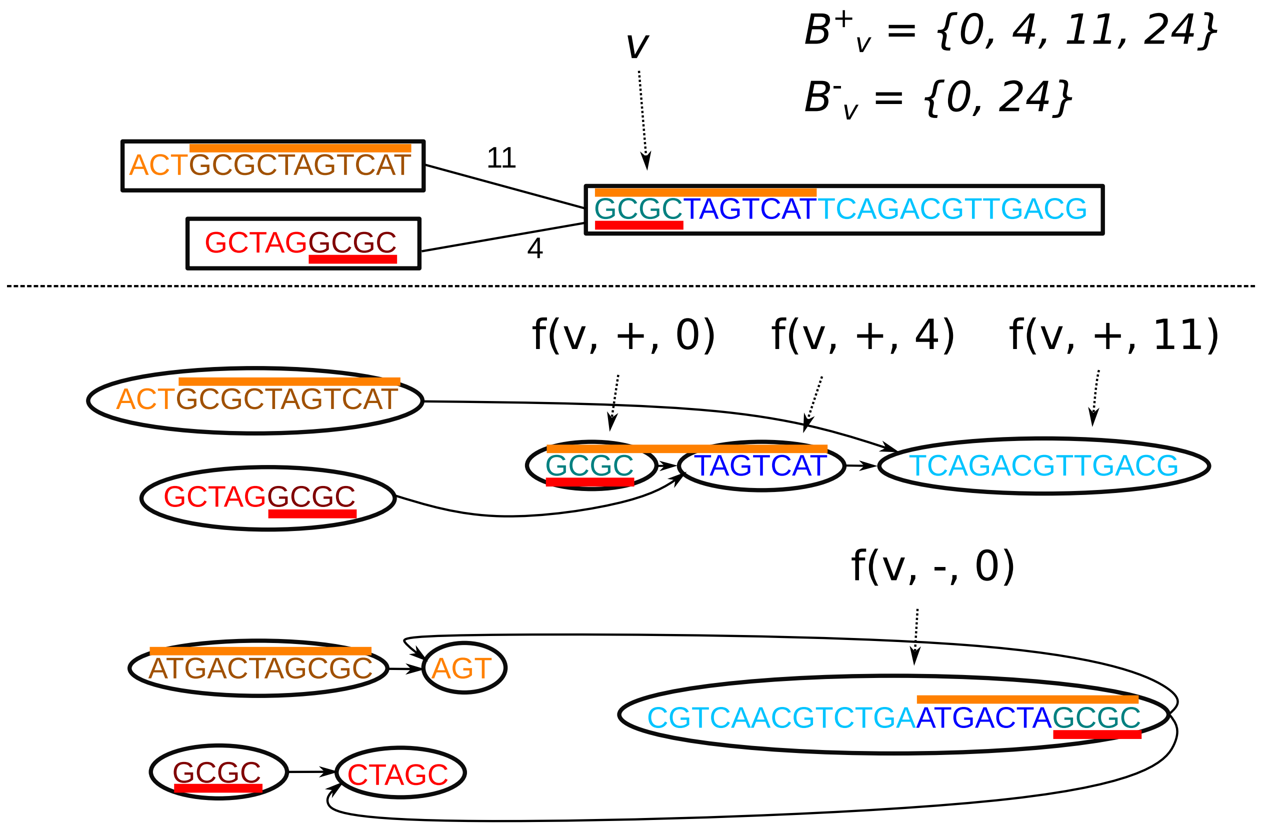


FIGURE 3.1 Converting a bidirected graph with variable exact edge overlaps to an alignment graph. Top: a bidirected graph with three nodes. The edges are labeled by their overlap. The red colored bars represent the same sequence, which should not be duplicated during traversal. Similarly, the orange colored bars represent the same sequence. Bottom: the alignment graph created from the top graph. The colors of the base pairs show how they match between the two graphs, with each sequence in the original graph represented by the same color in the alignment graph twice, once for the forward strand and once for the reverse complement. Similarly to the bidirected graph, the red and orange bars represent the same sequences. There are two subgraphs, one representing the forward traversal (top) and one the backward traversal (bottom) with reverse complemented node labels. Each edge introduces a breakpoint in the target node, splitting the node at the boundary of the overlap. The alignment graph then connects the ends of the overlap such that the overlapping sequence is only traversed once.

points $B_v^- = \{0, m_1, m_2, ..., |\sigma_b(v)|\}$. We also define a function $f : (V_b, o \in \{+, -\}, B_v^o) \rightarrow V_a$ which assigns each tuple of bidirected node, orientation and breakpoint position (except $|\sigma_b(v)|$) to one alignment graph node. Given the sorted sets of breakpoints, each successive pair of forward breakpoints $m, m' \in B_v^+$ causes a node to be inserted to the alignment graph with the label $\sigma_a(f(v, +, m)) = \sigma_b(v)[m, m')$ representing the forward traversal, and each successive pair of backward breakpoints $m, m' \in V_v^-$ adds one node with the label $\sigma_a(f(v, -, m)) = \overline{\sigma_b(v)}[m, m')$ representing the backward traversal. We also add edges from $f(v, +, m)$ to $f(v, +, m')$ and from $f(v, -, m)$ to $f(v, -, m')$. Then, each bidirected edge $e = (v_1, o_1, v_2, o_2, m)$ adds two edges to the alignment graph: one from $f(v_1, o_1, \max B_{v_1})$ to $f(v_2, o_2, m)$, and another from $f(v_2, \bar{o}_2, \max B_{v_2})$ to $f(v_1, \bar{o}_1, m)$. In addition to the breakpoints added by the edges, we also add a breakpoint every 64 base pairs to each node because this makes it easier to encode the alignment graph node sequences using 64-bit words.

A node in the bidirected graph with l nucleotides adds $2\lceil \frac{l}{64} \rceil$ nodes to the alignment graph, $\lceil \frac{l}{64} \rceil$ for the forward traversal and $\lceil \frac{l}{64} \rceil$ for the backward traversal, and each edge can split up to two nodes and add up to four edges in the alignment graph. The number of nucleotides in the alignment graph is exactly twice the number of nucleotides in the bidirected graph. Therefore the transformation produces an alignment graph whose size is within a constant factor of the bidirected graph.

During conversion, we also construct a *mapping* between the bidirected graph and the alignment graph. The mapping contains arrays $N : V_a \to V_b$, describing for each node in the alignment graph which node in the bidirected graph it was created from, $O : V_a \to \mathbb{N}$ describing the alignment graph node's offset within the bidirected node, and $D : V_a \to \{+, -\}$ describing the orientation of the alignment graph node within the bidirected node. Using these arrays, we define a function $pos : (V_a, \mathbb{N}) \to (V_b, \mathbb{N}, \{+, -\})$ which maps each base pair (encoded as a node and offset) in the alignment graph to a base pair and orientation in the bidirected graph as

$$pos(v, o) = \begin{cases} (N[v], O[v] + o, D[v]) & \text{if } D[v] = + \\ (N[v], |\sigma(N[v])| - (O[v] + o) - 1, D[v]) & \text{if } D[v] = - \end{cases}$$

Additionally, we store an array $A : V_b \to (V_a^n, V_a^n)$, mapping each bidirected node to the pair of alignment graph nodes which represent its forward and backward traversals.

Taken together, the tables described above define a bijection between base pairs in the alignment graph, and combinations of a base pair and orientation in the bidirected graph, enabling positions to be unambiguously converted between the two graph representations. Given the two graphs and the mapping, GraphAligner aligns the read to the alignment graph, and then converts the alignment back into the bidirected graph.

Both the read and the graph are allowed to contain ambiguous nucleotides (B, R, N, etc.) The alignment extension considers two ambiguous nucleotides a match if any of the possible nucleotides match; eg, R (A or G) matches W (A or T) because both of them could be A, but R (A or G) does not match Y (C or T) because there is no overlap. Only the non-ambiguous characters A, T, C and G are used for seeding.

3.3 Seed hit finding

Seeded alignment refers to a heuristic method where matches between two sequences are used to find an approximate alignment, which is then usually refined further with an exact dynamic programming algorithm. The matches can be exact or approximate matches of constant or variable length.

The first part of seeded alignment is finding seed hits. Here, we define seeds as exact matches

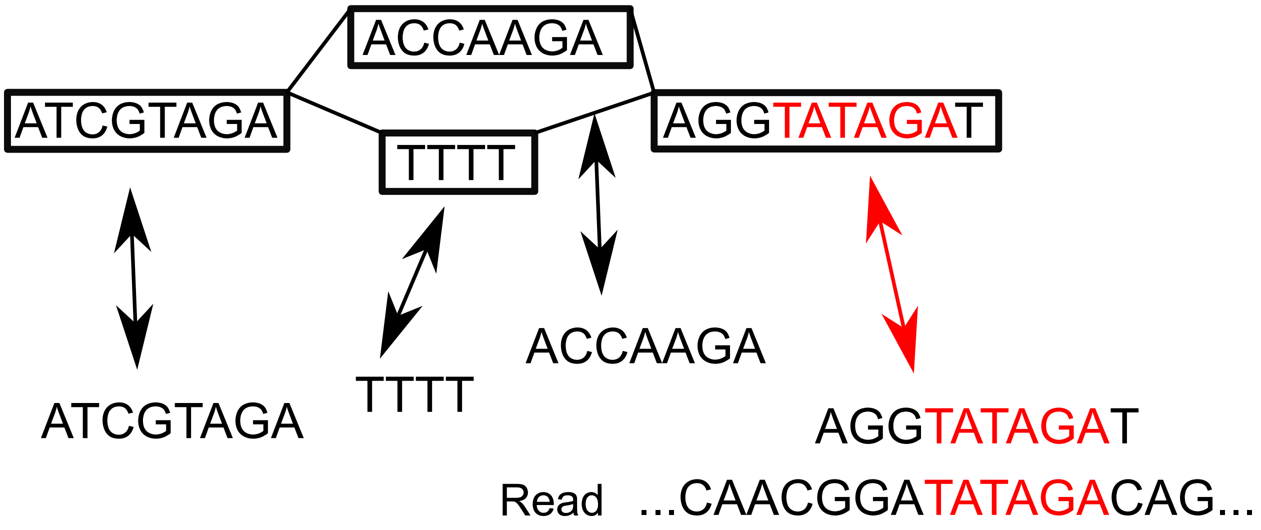


FIGURE 3.2 Seeding. Top: A graph with four nodes. Middle: The node sequences are extracted from the nodes. The arrows represent a mapping between the strings and nodes. Bottom: A read. Highlighted in red: Matches between the read and a string are converted into a match inside a node using the mapping.

between a read and a node sequence, but other definitions exist in the literature. Methods for finding exact matches between a read and paths in a graph have been developed [89–91, 108]. Finding matches between a sequence and a graph is difficult due to the exponential number of paths. GraphAligner uses a simple method for transforming text matching in graphs to text matching in strings. Instead of matching reads to paths in the graph, reads are matched to node sequences in the graph. The nodes can be treated as a collection of strings which enables using efficient string matching algorithms. Reverse complement matches are also allowed. Figure 3.2 shows an example of matching a read to nodes in a graph. Note that we use the node sequences from the original bidirected graph, not from the directed alignment graph. The matching position is then converted from the bidirected graph to the alignment graph.

This approach finds seed hits which are entirely contained in a node. However, seed hits which cross two or more nodes are not found. The experiment in Section 3.9.2 shows that this works in practice for long reads, since they almost always cross a long node. However, for short reads, a graph can have variant rich areas where the reads will systematically fail to align due to the short length of the nodes. In the special case of de Bruijn graphs all seed hits of length up to k are found.

GraphAligner's default method uses minimizers [41] for finding matches. A hash function f assigns a hash score to each k-mer. A window of length $w \geq k$ is slid through a sequence. Each k-mer in the window is hashed with f, and the one with the smallest hash value is selected as the minimizer for that window. The selected k-mers are called minimizers. The minimizers are then stored in a index. The index uses succinct data structrures and is implemented using SDSL [109].

Building the minimizer index from a graph is multithreaded. Given n threads, each thread picks nodes one at a time, and finds the minimizers in that node. The minimizers store the k-mer, the node ID and the position within the node. The threads divide the minimizers into n buckets, implemented as parallel queues, based on the modulo of their k-mer. After all nodes have been processed, each thread picks one bucket and builds a bucket index from it. The minimizers in the bucket are first sorted based on their k-mer. Then, a bitvector representing different k-mers is built. The bitvector is set at indices where the current k-mer is different from the previous k-mer. A rank-select structure is built from the bitvector. Then, a minimal perfect hash function [110] is built to assign each k-mers to the rank of the k-mer in the bitvector. Figure 3.3 shows the pipeline for indexing a graph.

To query a k-mer, first the appropriate bucket index is found using the modulo of the k-mer. Then, the minimal perfect hash function is used to query the rank of the k-mer. The rank-select structure is then used to find the index in the sorted array where the k-mer is stored. Since the minimal perfect hash function can produce false positive hits for k-mers which were not used in constructing it [110], existence of the k-mer is verified by checking that the stored k-mer is equal to the queried k-mer. The number of occurrences of the k-mer can be checked in constant time by querying the index of the next-ranked bit. To retrieve the positions, the sorted k-mer array is iterated at the appropriate range.

When finding seed hits, first a maximum number of seeds is calculated using a *seed density* parameter d. All k-mers of the read are queried to find matches and their frequencies. Given a read of length l and the *seed density* parameter d, only the least frequent ld minimizer hits are kept. In case of ties, all minimizers with frequency equal to the ld'th minimizer are kept.

The default values use $k = 19$, $w = 30$, $d = 5$. These values are tuned for aligning reads to de Bruijn graphs. We have noticed that good parameters for aligning reads to a de Bruijn graph lead to poor alignment quality on variation graphs, and good parameters on variation graphs lead to high runtimes on de Bruijn graphs without improving alignment quality. For variation graphs, we instead recommend the parameters $k = 15$, $w = 20$, $d = 10$, which are the parameters used in the linear comparison experiment, variant graph experiment and comparison to vg experiment.

In addition to minimizer-based seeding, GraphAligner also contains modes for seeding using maximal exact matches (MEMs) and maximal unique matches (MUMs) [43]. Maximal exact matches are matches between two strings such that extending the match by one character in either direction would remove some matches. Maximal unique matches are MEMs which have exactly one match in both strings. Like minimizer matching, MEM/MUM matching also matches only to the node sequences and not to paths in the graph. MEM/MUM seeding uses the MUMmer4 library [27] for indexing and finding matches.

The default seeding mode is minimizers with $k = 19$, $w = 30$, $n = 5$ and chunk length $c = 100$. These values are tuned for aligning reads to de Bruijn graphs. We have noticed that good parameters for aligning reads to a de Bruijn graph lead to poor alignment quality on variation graphs, and good parameters on variation graphs lead to high runtimes on de Bruijn graphs without improving alignment quality. For variation graphs, we instead recommend the parameters $k = 15$, $w = 20$, $d = 10$, which are the parameters used in the linear comparison experiment, variant graph

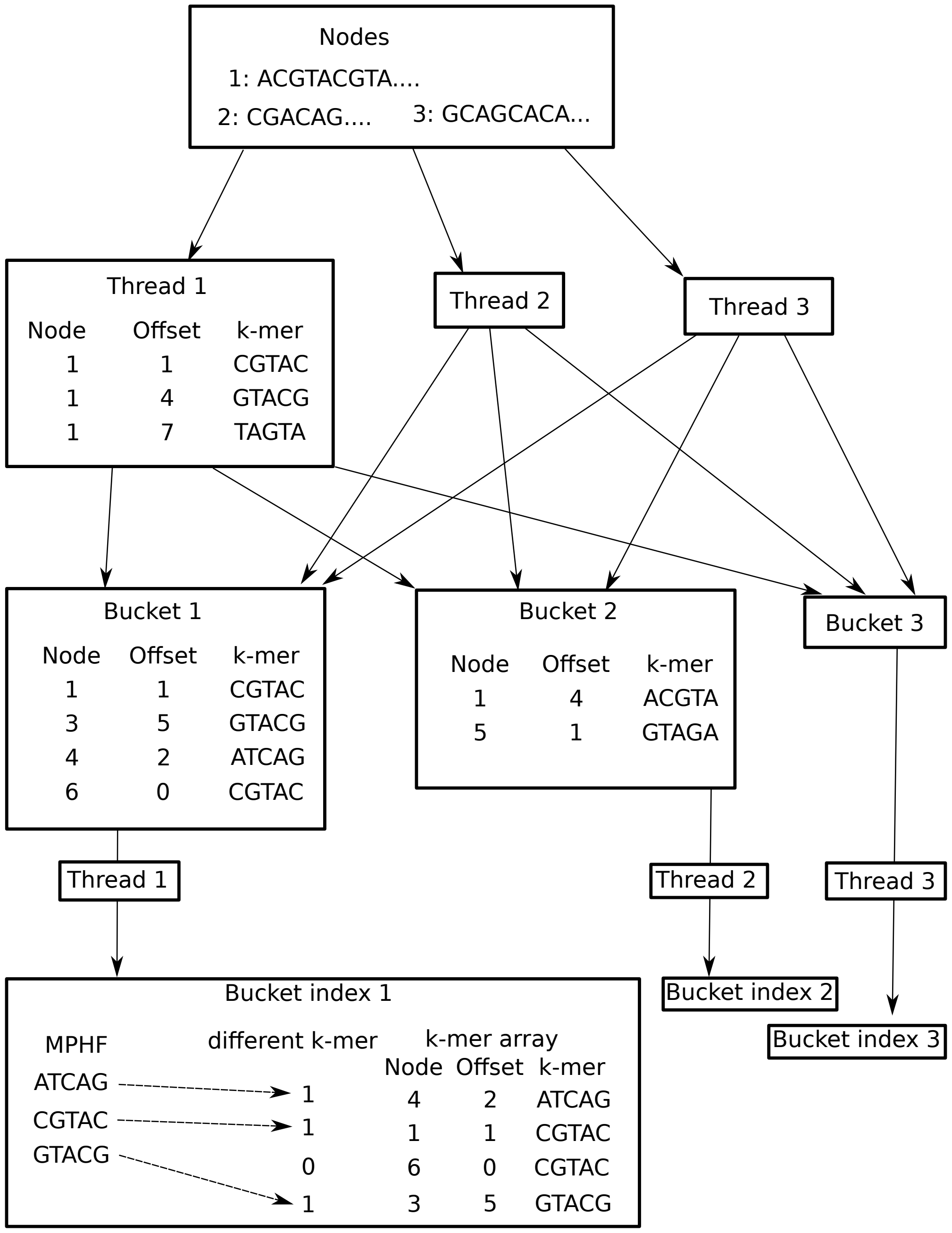


FIGURE 3.3 Building a minimizer index from a graph. Only the nodes of the graph are considered when building the index, and edges are ignored. Each node has an ID and a sequence. At start all nodes are unprocessed. Threads pick nodes one at a time from the pool of unprocessed nodes, and find minimizers in the node sequence. Then, the threads distribute the minimizers into buckets according to the modulo of their k-mer. Once all nodes have been processed, the threads proceed to index the buckets. Each thread picks one bucket and indexes it into a bucket index. The bucket index contains an array of the minimizers in that bucket sorted by the k-mer, a bitvector representing indices where a k-mer is different from the previous one, and a minimal perfect hash function which assigns each k-mer to the rank of the bit which represents the first instance of that k-mer in the sorted array.

experiment and comparison to vg experiment. The MEM/MUM mode is not used by default as it is slower than minimizer seeding, and is included due to being implemented before minimizer based seeding with no reason to remove it. In addition to the two built-in seeding methods, arbitrary seeds from an external method can be used. In this case the seeds must be inputed in vg's GAM format.

Finally, reads may be aligned without seeding at all. In this case, the bit-parallel DP algorithm is applied to the topmost 64 rows of the DP table. The runtime of this mode scales to the size of the graph so it not appropriate for large graphs.

3.4 Seed hit clustering

Typical alignment approaches [29] *chain* seeds to find the approximate position of the alignments. For linear sequences, seed chaining is solved with the *co-linear chaining* problem that exploits the fact that calculating the distance between seeds in a linear sequence is trivial. However, for graphs, the distance between seeds can be ambiguous as there are multiple paths connecting the seeds, and finding the distance in a graph is computationally more expensive than in a linear sequence [111].

GraphAligner clusters seed hits within chains of superbubbles. We use the algorithm from Onodera et al. [73] to detect superbubbles. Note that the superbubbles are found from the directed alignment graph, not from the original bidirected graph.

Given two superbubbles, we say that they belong in the same *chain* if the end node of one superbubble is also the start node of the other. Superbubbles may be chained this way to longer chains and we say that they form a *chain of superbubbles*. In addition to superbubbles, we treat tips and small cycles as special cases that are included in the chain of superbubbles. An important property of a chain of superbubbles is that they induce an acyclic subgraph. The nodes can therefore be assigned linearized positions. GraphAligner arbitrarily picks one node in the chain of bubbles as the start node, and then performs a breadth-first search along the chain to assign a linear position to each node. The pseudocode for the linearization is in Algorithm 12.

Given a chain of superbubbles, we can assign all seed hits in the chain a linear position. A position at offset o of node v is assigned a linearized position $D[v] + o$. Then, chaining algorithms for linear sequence alignment can be used for chaining the seed hits. We use the seed clustering algorithm from minimap [47], not to be confused with the seed chaining algorithm from minimap2 [29], to assign seed hits to clusters. Here we briefly recap the seed clustering algorithm from minimap. Given a seed hit with position r in the read and a linearized position b in the chain of superbubbles, define the *diagonal position* of the seed hit as $d = r - b$. Then, two seeds in the same chain of superbubbles whose diagonal positions d_1 and d_2 are within a cutoff $c = 100$, that is, $|d_1 - d_2| \leq c$, are connected together. The transitive closure of the connected seeds is the cluster.

Then, seed hits are scored according to their cluster size and uniqueness, with matches that occur fewer times in the graph weighted higher. Given a seed hit whose sequence occurs x times

Algorithm 12 Assigning linear positions to nodes in a chain of superbubbles

```
1: Input: an alignment graph $(V, E, \sigma)$ and a set of nodes $C$
2: Output: array $D$ with the linearized distance for each node in $C$
3: set all nodes in $C$ as unvisited
4: $S \leftarrow$ an empty stack
5: $S.push($an arbitrary node from $C, 0)$
6: while $S$ is not empty do
7:     $(v, d) \leftarrow S.pop()$
8:     if $v$ has been visited or $v \notin C$ then
9:         continue
10:    end if
11:    $D[v] \leftarrow d$
12:    set $v$ to visited
13:    for $u : (v, u) \in E$ do                          ▷ out-neighbors of $v$
14:        $S.push(u, d + (|\sigma_u|))$
15:    end for
16:    for $u : (u, v) \in E$ do                          ▷ in-neighbors of $v$
17:        $S.push(u, d - (|\sigma_v|))$
18:    end for
19: end while
```

in the graph, and a maximum occurrence m, the unclustered *score* of the seed hit i is $s'_i = m - x$. Then, given a cluster C, we calculate the number of base pairs in the read covered by at least one seed c_C. The score of a seed hit that belongs in cluster C will then be $s_i = s'_i + c_C$.

The seed hits are ordered based on their clustered scores and extended from best scoring to worst scoring. Since the seed hits are not clustered arbitrarily across the graph, but only in simple subgraphs, the seed hit clustering is not used for limiting the paths explored or deciding when to end the alignment. The alignment algorithm used for the extension step instead decides which paths to explore and when to end the alignment. Seeds included in alignments from previously explored seeds are skipped.

Finally, a *seed extension density* e parameter is used for choosing how many seed hits to extend. Given a read of length l and the extension density parameter e, seeds are extended starting from the highest scoring seed until le seed hits have been extended, with ties also extended. Seeds which are skipped due to being included in a previous alignment do not count against this limit. This filter is applied after the seed hits have been clustered and scored. The default values for e is $e = 0.002$ for de Bruijn graphs and $e = 1$ for variation graphs.

3.5 Banded alignment on graphs

Banded alignment on sequence-to-sequence alignment refers to methods that only calculate a part of the DP table [19, 25]. Usually some seeding algorithm is used to find an approximate alignment and then a parallelogram shaped area of the DP table containing the approximate alignment

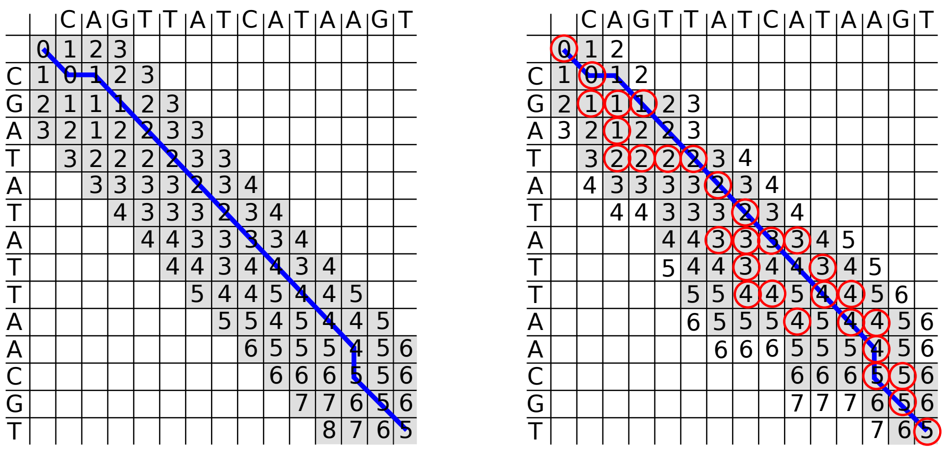


FIGURE 3.4 Left: regular banded alignment with $b = 3$. The reference is on top and the query on the left. The gray cells are inside the band and are calculated. The blue line shows the traceback of the optimal alignment. Right: score based banding with $b = 1$. The reference is on top and the query on the left. The gray cells are inside the band and the blue line is the traceback. The red circled cells are the minimum for each row, which are discovered during the calculation of the matrix and define whether a cell is inside the band or not; a cell is inside the band if its score is within b of the minimum score in the same row. The cells with a number on a white background are calculated to discover the end of the band, but they are not inside the band and are ignored when calculating the next row. The band can wander around the DP matrix and change size, automatically spreading wider in high error areas and narrower in low error areas. Note that the score based banding parameter is 1 in comparison to 3 in the regular banding to the left. The implementation uses a coarser band of 64 x 64 blocks instead of individual cells.

is calculated. This approach cannot be used in graphs since the band could grow very large, and the overhead of keeping track of the parallelogram could grow exponentially.

Recently a dynamic banding approach was proposed for linear sequence alignment [26]. The approach allows the band to move during the alignment based on the scores of the alignment. The method requires calculating the DP matrix in an antidiagonal order, which cannot be easily extended to graph alignment since the antidiagonal is ambiguous for forks.

GraphAligner uses a dynamic banding approach based on alignment scores. Instead of defining the band based on the location in the DP table, it is defined based on the scores of the cells in the DP table. The idea is that each row has a minimum score, which is the best possible alignment ending at that row, and cells whose score is within a banding parameter b to the minimum score are kept in the band. Since the scores are not known before alignment, the band is not known either. The minimum scores and the band are instead discovered during calculation. This requires calculating the DP table beyond the band to discover the border of the band as well. Figure 3.4 shows an example of dynamic score-based banding.

This approach handles arbitrary graph topologies and implicitly limits exploration of alternate paths. When two paths have similarly good alignments, the scores will be similar and both paths will be inside the band. If on the other hand one of the paths aligns poorly to the read, the scores along that path will be poor and the path will eventually fall out of the band. Figure 3.5 shows how the score-based banding limits exploration of alternate paths. The band essentially aligns the read to different paths and dynamically stops the alignment once it becomes clear which path the read truly aligns to. An important feature is that score-based banding does not need to explicitly handle any kind of topological features of the graph. Instead this behavious automatically follows from the definition of the score-based banding.

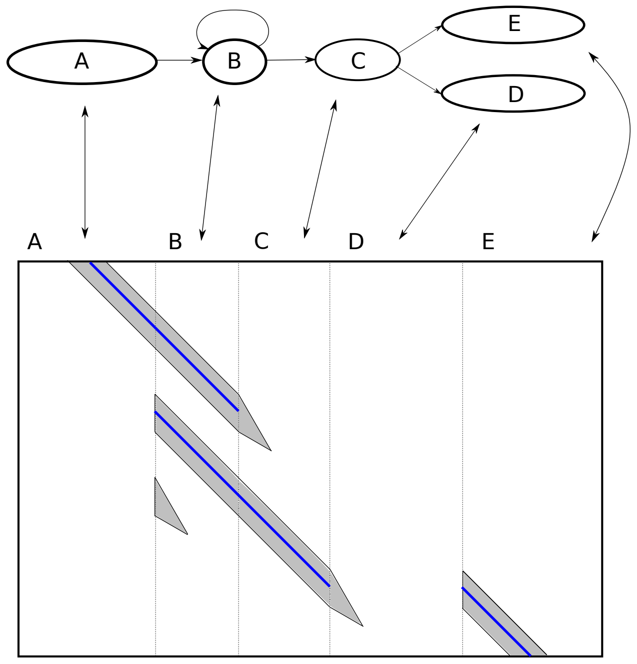


FIGURE 3.5 Dynamic score-based banding applied on a graph. Top: an alignment graph. Bottom: The DP matrix for aligning a read to the above graph. The arrows show the correspondence between nodes in the graph and columns in the DP matrix. The dotted lines separate the nodes. The gray area represents parts of the DP matrix which are calculated, and the parts in the white area are not calculated. At each fork, the band spreads to all out-neighbors. The score-based banding implicitly limits the exploration of the alternate paths; as the scores in the alternate paths become worse than the optimal path, the explored part shrinks until finally the exploration stops completely. The blue line shows the backtrace path.

Due to the bitvector based extension algorithm, the banding in GraphAligner is slightly different from the theoretical description above. The band does not act on individual cells, instead on blocks of 64 rows and up to 64 columns, where the columns of each block correspond to one node in the chopped alignment graph. The scores are not compared on each row, instead the scores along the bottom row of the block are used as the minimum scores. Scores at the rightmost column of a block are used for checking whether the block is inside the band, but are not used for defining the minimum score. Figure 3.6 illustrates the banding with blocks. Note that the block-based banding contains all cells which would have been included in the cell-based banding.

Since b represents a score difference, the score-based banding trivially keeps the guarantee that as long as the number of mismatches is at most b, the optimal alignment is found. However, score-based banding does not place any guarantees on the size of the band. In the worst case, an arbitrarily small b can still contain an arbitrarily large graph. For example, a fully connected graph will always

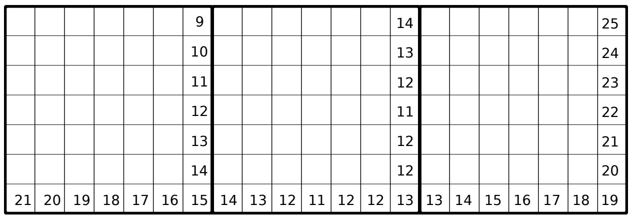


FIGURE 3.6 Block-based band. The three solid squares represent 7×7 blocks, over which the band is defined. The blank cells are ignored when determining whether a block is inside the band or not. The scores in the bottom row define the minimum score, which is 11. The scores in the rightmost columns of the blocks are not used for the minimum score. The scores in the bottom row and the rightmost column of each block are compared with the minimum score to determine whether the block is inside the band. Given a banding parameter $b = 2$, all blocks which have a cell with a score of at most 13 are within the band. The leftmost block is inside the band since there is a cell on the rightmost column with a score of 9. Similarly, the rightmost block is inside the band since there is a cell its the bottom row with a score of 13. In practice, the blocks are 64 rows high and up to 64 columns wide.

be entirely contained in the band even with $b = 1$. Degenerate cases such as fully connected graphs are unlikely to occur in practice but de Bruijn graphs can have highly tangled subgraphs. Figure 3.7 shows an example of a very tangled subgraph.

GraphAligner also includes a second parameter for banding, the *tangle effort*. Tangle effort limits the amount of computation on very tangled regions of the graph. Given a tangle effort parameter C, the extension algorithm keeps track of how many columns have been calculated. Once the number of calculated columns in the current 64-row slice of the DP table grows above C, the current slice is kept as is and the algorithm moves on to the next slice. This can cause inoptimal alignments or even failed alignment in tangled regions.

The bit-parallel extension algorithm described in earlier sections uses the *minimum changed value* to define an order for computing the columns. If the tangle effort parameter is not used, the extension algorithm uses the minimum changed value exactly as described earlier. However, if the tangle effort is used, a different ordering is used, the *minimum changed priority value*. Given an observed error rate e, the *priority value* of a cell on row m with an edit distance k is $\frac{k}{e} - m$, or $64k - m$ if $e \leq \frac{1}{64}$. When recalculating a column, the *changed priority value* of a cell is the priority value of the cell if it changed, and infinite for those which did not change. The *minimum changed priority value* of a column is the minimum of the changed priority values of the cells in the column. The minimum changed priority value essentially describes "how good" an alignment at a cell is. A zero is as good as the current best alignment, a positive value is worse, and negative is better. The minimum changed priority value is essentially a greedy heuristic for exploring the most promising paths first. Using the minimum changed priority value leads to a higher probability of finding the optimal alignment early and successfully aligning through a tangle. When calculating the full DP table, using the minimum changed priority value would lead to the DP table eventually converging to the correct scores, but the worst case runtime can be worse than the minimum changed value.

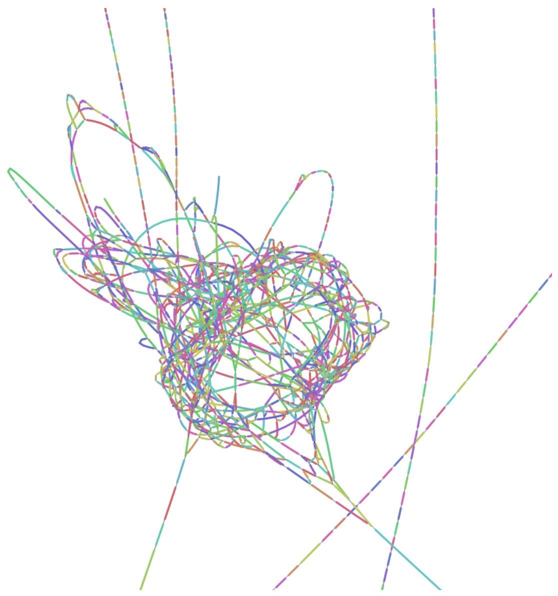


FIGURE 3.7 A tangled subgraph of a whole human genome de Bruijn graph. The subgraph includes nodes which are traversed by the alignment of one read originating from the MHC region to a whole human genome de Bruijn graph. The picture is zoomed in to one region of the subgraph and does not contain all of the nodes. Visualized with Bandage [98].

3.6 Storing a sparse DP matrix

In sequence-to-sequence banded alignment the DP table can be efficiently stored as $2b$ diagonal. However in sequence-to-graph alignment the banded table cannot be stored contiguously due to the non-linear nature of graphs. We treat the DP table as a three-dimensional matrix, with one dimension for node ID, one for node offset and one for sequence position.

The implementation stores the DP table as a hash table from a node ID to a sparse representation of the table. The sparse representation explicitly stores the bottom-left and bottom-right "corner" cells. The score differences along the bottom row and the leftmost and rightmost columns are also stored. Figure 3.8 shows an example of this. The middle cells are not stored. Instead, they are recalculated when necessary. This only happens when recalculating a cycle, in which case they would have to be recalculated anyway, and when backtracing the alignment, which only requires calculating the path taken by the backtrace. The sparse representation stores the data in the same way as the bitvector operations, so there is no runtime overhead from converting between formats or from compression. The sparse representation uses 56 bytes per node, plus memory overhead from the hash table. For comparison, the information theoretical lower bound for storing all cells in the DP table for one node with optimal compression is $\frac{\log_2 3^{64*64}}{8} \approx 812$ bytes, and storing only the border cells is $\frac{\log_2 3^{64+64+62}}{8} \approx 38$ bytes

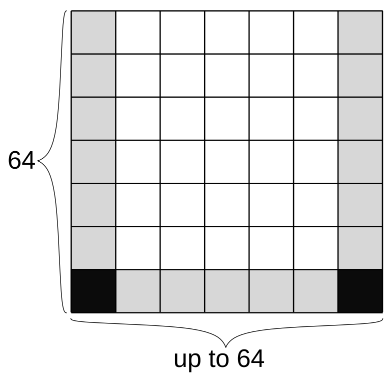


FIGURE 3.8 Sparse storage of the DP matrix. Each node is stored in blocks of 64 rows and up to 64 columns. The scores of the corner cells (solid black) are stored explicitly, using 4 bytes per cell. The border cells (gray) are stored with a score difference, using 2 bits per cell. The middle cells (white) are not stored, and are recalculated when needed.

3.7 Partial alignments

Previously software such as BLAST [37] have used the *X-drop* heuristic [112] to end alignment. In the X-drop heuristic, the algorithm keeps track of the highest alignment score seen so far. Once the scores within the current row to be calculated drop below a cutoff defined on the highest alignment score and a parameter, the alignment is ended and the cell with the highest alignment score is used to start the backtrace. The X-drop heuristic requires using local alignment with a scoring scheme where higher values are better alignments, and it is not trivial to correctly extend it to the unit costs required by the bit-parallel algorithm.

During alignment, we use Viterbi's algorithm [113] to estimate the correctness at each slice boundary. That is, we seek to estimate the probability that the slice contains the correct alignment. The observations of the algorithm are the minimum scores at the end of each slice. Conceptually, we use a hidden Markov model with two hidden states, which are labeled *"correctly aligned"* and *"wrongly aligned"*. We model the emissions and transition probabilities such that the correctly aligned state outputs an error rate of 20% and the wrongly aligned an error rate of 50%. These error rates were selected empirically by aligning Oxford Nanopore (ONT) reads to either the correct or the wrong genomic position, using the assumption that the errors of reads to be processed are at most as high as for these ONT reads. The probabilities of the correct and wrong states and their predecessor states are calculated for each slice during alignment. After calculating slice $n+1$, we define slice n as *guaranteed correct* if the predecessor of the wrong state in slice $n+1$ is the correct state in slice n. The intuition behind this is that any alignment in slices $n+1$ and later, correct or wrong, must backtrace through the correct state at slice n so the read is correctly aligned at least until that point. We also similarly define a slice n as *guaranteed wrong* when predecessor for the correct state in slice $n+1$ is the wrong state in slice n. Figure 3.9 shows an example of this.

We use the correctness estimate to vary the banding parameters. We use two parameters, an

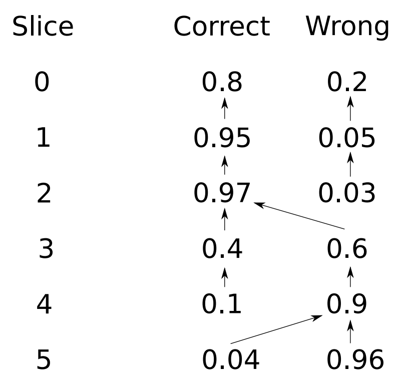


FIGURE 3.9 An example of using Viterbi's algorithm for estimating correctness per slice. The error rates of the minimum alignment per slice (not shown in figure) are the observations. The numbers represent the probability of the alignment being in the specific state at the specific slice. The arrows represent the predecessor state for each state in each slice. Slice 2 is guaranteed correct since the predecessor for the wrong state in slice 3 is through the correct state. Similarly, slice 4 is guaranteed wrong since the predecessor for the correct state in slice 5 is through the wrong state. None of the other slices are guaranteed correct or wrong. The final backtrace will consider slices 0, 1 and 2 correctly aligned and slices 3 to 5 wrongly aligned, and only the sequence in slices 0-2 will be reported in the alignment.

initial banding parameter b and a ramp banding parameter $B > b$. Once the probability of the wrongly aligned state is higher than the probability of the correctly aligned state, we backtrace to the last guaranteed correct slice, switch to the higher ramp banding parameter, and re-align until we have reached the original slice. Note that this is a looser condition than reaching a guaranteed wrong slice.

We also use the Viterbi estimate to end the alignment. Once we have reached a guaranteed wrong slice, the extension can no longer produce anything useful. In this case, we backtrace to the last correct slice and return the partial alignment of the read up to that position.

After extending the seed hits, we are left with a list of partial alignments. We then select a non-overlapping subset of primary and supplementary alignments in a heuristic manner. We greedily pick alignments from longest to shortest, and include an alignment as long as it does not overlap with a previously picked alignment. The primary and supplementary alignments are then written as output. The overlapping alignments are considered secondary and discarded by default, with an optional switch to output secondary alignments as well.

3.8 Parallelism

GraphAligner is trivially parallelized by giving each read to a separate thread for aligning. The alignment of individual reads is not parallelized. One IO thread reads sequences from a file and passes them to a lock-free single producer multiple consumers queue. Worker threads pick one read from the queue, align it, and pass the resulting alignment to an another lock-free multiple producer single consumer queue. A second IO thread takes the alignments from the queue and writes them

Aligner	Reads correctly aligned	CPU-time (HH:mm:ss)	Peak memory (Gb)
minimap2	95.1%	44:26:58	20.0
GraphAligner	95.0%	127:16:34	72.1

TABLE 3.1 Results of the linear comparison experiment. Simulated reads were aligned to the GRCh38 reference genome with minimap2 and GraphAligner.

to a file. The worker threads then free the memory used by the alignment.

3.9 Experiments

We performed five experiments involving GraphAligner. First, GraphAligner was compared to the well-optimized linear aligner minimap2 [29]. Second, GraphAligner's mapping accuracy on a variant graph was tested with reads of different lengths to measure the effect of the heuristic seeding approach. Third, GraphAligner was compared to the state of the art graph aligner vg [1]. Fourth, we built a simple genotyping pipeline using GraphAligner and vg. Fifth, an error correction pipeline using Graphaligner was compared to the state of the art error correctors.

3.9.1 Comparison to linear aligners

Regular sequence-to-sequence alignment is a special case of sequence-to-graph alignment, where the graph consists of a linear chain of nodes. We compare GraphAligner to a well-optimized sequence-to-sequence aligner, minimap2 [29], in whole human genome read alignment. We simulated 20x coverage reads from the GRCh38 reference using pbsim [99] with default parameters. We filtered out reads shorter than 1000bp and reads containing any non-ATCG characters. Then, we aligned the reads to the reference using both minimap2 and GraphAligner. Then, we evaluated the mapping accuracy. We adopt the criteria used in the minimap2 evaluation [29] and consider a read correctly mapped if its longest alignment overlaps at least 10% with the genomic position from where it was simulated.

Table 3.1 shows the results. GraphAligner and minimap2 both align approximately as accurately, with minimap2 aligning slightly more reads correctly (95.0% vs 95.1%). GraphAligner takes about 3$\times$ the runtime of minimap2, which we consider to be a modest overhead for a tool able to handle graphs in comparison to a highly optimized sequence-to-sequence mapping tool. Note that minimap2 is faster than commonly used competing tools, such as BWA-MEM [28], by more than one order of magnitude [29].

3.9.2 Aligning to a graph with variants

In this experiment we evaluated the mapping accuracy to a graph with variants. We used the chromosome 22 reference (GRCh37) and all variants in the Thousand Genomes project phase 3 release [114]. We constructed a variation graph from the reference and the variants using vg [1], producing a graph of chromosome 22 with 2,212,133 variants, containing on average one variant every 15 base pairs in the non-telomeric regions (the *variant graph*). Then, we simulated reads of varying lengths using pbsim [99] and aligned them to the graph with GraphAligner. We consider a read correctly mapped if its longest alignment overlaps at least 10% with the genomic position from where it was simulated and evaluate the number of reads correctly aligned. We also aligned the same reads to the chromosome 22 reference without variants (the *linear graph*) with GraphAligner to differentiate between reads which could not be aligned due to variants, and reads which could not be aligned due to other reasons such as short read lengths leading to missed seeds.

Figure 3.10 shows the results. For comparison purposes, the blue curve represents the results from mapping reads simulated from GRCh38 back to the (linear) reference genome and hence indicate the performance that can be achieved in an idealized setting. When aligning to the variant graph, 95% of the reads are correctly aligned once read length grows above 1,200 base pairs. At 1,500 base pairs, 97.0% of the reads are correctly aligned to the variant graph. The results show that GraphAligner is capable of aligning long reads accurately to a variation-rich graph.

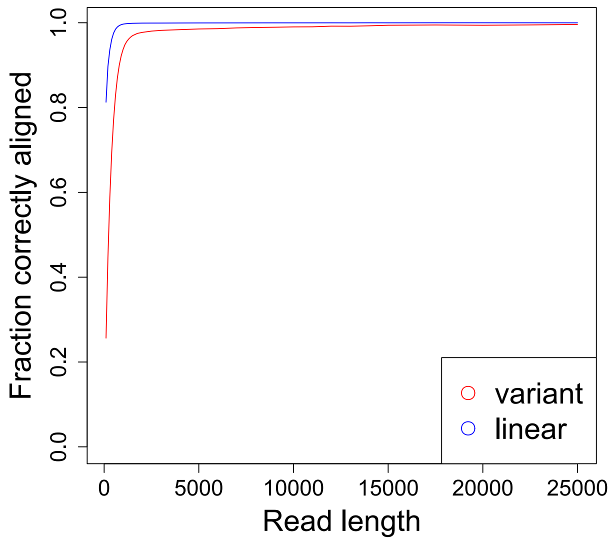


FIGURE 3.10 Fraction of reads correctly aligned at varying read lengths for the SNP graph and the linear graph.

Aligner	Reads correctly aligned	CPU-time (HH:mm:ss)	Peak memory (Gb)
Original	-	-	-
vg index	-	1:07:44	12.1
vg map	93.8%	3:13:15	4.1
GraphAligner	96.6%	0:19:30	3.6

TABLE 3.2 Results of the comparison to vg. Simulated reads were aligned to a chromosome 22 variation graph using both GraphAligner and vg.

3.9.3 Comparison to vg

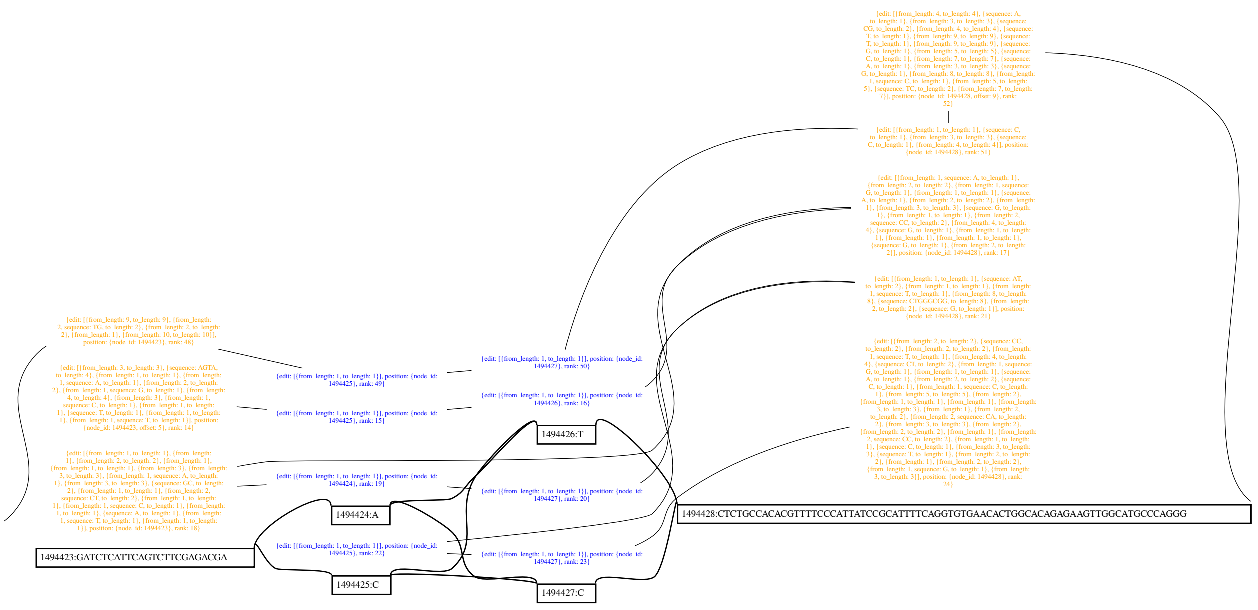
In this experiment we compared using GraphAligner and vg [1] to align long reads. We used the graph from the previous experiment containing the chromosome 22 reference and all variants in the Thousand Genomes project phase 3 release [114]. We simulated reads from the chromosome 22 reference using pbsim [99] with default parameters. Then we aligned the simulated reads to the graph using GraphAligner and vg.

Table 3.2 shows the results. GraphAligner aligned 96.6% of reads correctly, which is consistent with the results of the variation graph experiment. In contrast, vg aligned 93.8% of reads into the correct genomic region. However, we found that some of the alignments by vg were not consistent with graph topology, that is, the alignment traversed through nodes which are not connected by an edge. In some cases the alignment "looped back" into the same reference area multiple times and even covered both alleles of a variant. Figure 3.11 shows an example of a vg alignment which is not a path. We did not evaluate how many of vg's alignments were inconsistent with graph topology. GraphAligner's runtime and peak memory includes both indexing and alignment. Despite including the indexing phase, we see that GraphAligner is almost ten times faster than vg's mapping phase. When including vg's indexing as well, GraphAligner is over thirteen times faster than vg. Peak memory use is three times smaller.

3.9.4 Variant genotyping

We implemented a simple variant genotyping pipeline for long reads. First, a list of reference variants and a reference genome are used to build a pangenome graph using vg [1]. Then, long reads are aligned to the pangenome graph with GraphAligner. Finally, vg is used to genotype the variants according to the long read alignments.

We tested our variant genotyping pipeline using 35x coverage PacBio hifi reads from the individual HG002 [115], using the Genome in a Bottle (GIAB) benchmarking variant set version 3.3.2 for GRCh38 [116] as the ground truth. We tested three different scenarios: first, an ideal scenario where we use the variants in the GIAB variant set to build the graph; second, a more realistic scenario where we used variants from a different source, using the variant set by Lowy-Gallego et



1494423:GATCTCATTCAGTCTTCGAGACGA
1494424:A
1494425:C
1494426:T
1494427:C
1494428:CTCTGCCACACGTTTTCCCATTATCCGCATTTTCAGGTGTGAACACTGGCACAGAGAAGTTGGCATGCCCAGGG
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494425}, rank: 49}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494425}, rank: 15}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494424}, rank: 19}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494425}, rank: 22}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494427}, rank: 50}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494426}, rank: 16}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494427}, rank: 20}
{edit: [{from_length: 1, to_length: 1}], position: {node_id: 1494427}, rank: 23}

FIGURE 3.11 An alignment produced by vg [1] which is inconsistent with graph topology. A read simulated by PBSIM [99] was aligned with vg to a variant graph of the chromosome 22 created by vg. The alignment was then visualized with vg, and manually cropped to a position containing two consecutive SNPs. The solid boxes represent nodes in the graph, and the thick black lines between the corners are edges in the graph. There are two SNPs adjacent to each other, the first with alleles A and C, and the second with alleles T and C. The alignment is represented by the blue and yellow texts connected by the thin black lines. The alignment passes through both branches of both SNPs, in fact covering the allele A in the leftmost SNP three times, and covers the flanking regions multiple times as well. Since the graph is acyclic, this alignment cannot be consistent with any path in the graph.

al. [117] called from the GRCh38 genome using the data from phase 3 of the Thousand Genomes Project (1000G) to build the graph; and third, using the variants from 1000G to build the graph but only evaluating the accuracy on variants which occur in both the 1000G and the GIAB variant set (1000G+GIAB). The reason for using the three different scenarios is that the genotyping pipeline cannot call novel variants, instead it only genotypes variants which are already in the list of reference variants. This separates errors caused by the pangenome approach, and errors caused by imperfect reference variant set; the GIAB scenario will show how the pipeline would behave if the reference variant set was perfect, while the 1000G scenario will show the performance with a realistic, imperfect reference variant set and the 1000G+GIAB scenario will show the performance of in a realistic setting for those variants that the pipeline could in principle genotype.

We evaluated the genotyping accuracy using RTG Tools vcfeval [118], which computes precision and recall for all variants, SNPs only and non-SNPs only. vg produces a confidence for each variant, and the evaluation produces a precision-recall curve for different confidence thresholds. We selected the threshold with the highest F-measure and report the precision and recall for that threshold. We evaluated the results in the Genome in a Bottle high confidence regions from all chromosomes in each scenarios.

Table 3.3 shows the results. The genotyping accuracy is high in the GIAB scenario, but lower

Scenario	Variants	Precision	Recall	F-measure
GIAB	all	0.9929	0.9774	0.9851
	SNP	0.9994	0.9840	0.9916
	non-SNP	0.9518	0.9353	0.9435
1000G	all	0.9694	0.8806	0.9229
	SNP	0.9806	0.9352	0.9574
	non-SNP	0.8462	0.5417	0.6606
1000G+GIAB	all	0.9685	0.9712	0.9699
	SNP	0.9801	0.9797	0.9799
	non-SNP	0.8556	0.8893	0.8721

TABLE 3.3 Results of the variant genotyping experiment.

in the 1000G scenario. This shows that the choice of variant set affects the accuracy noticably with the F-measure dropping from 0.985 to 0.930. However, when excluding variants that the pipeline could not genotype even in principle, the F-measure is 0.970. This shows that a large part of the missing recall in the 1000G scenario is from variants that are not included in the reference variant set.

Although previous publications [115] have shown performance exceeding the results in Table 3.3, the genotyping experiment shows an example use case for GraphAligner. The major limitation of the pipeline is that it cannot call novel variants, instead only genotyping known variants. We did not try varying the parameters of vg's genotyping module or otherwise adjusting the genotyping process, which is tuned for short read genotyping and may not be optimal for long reads.

3.9.5 Error correction

We have implemented a hybrid error correction pipeline based on sequence-to-graph alignment. Aligning reads to a de Bruijn graph (DBG) is a method of error correcting long reads from short reads [2, 119]. The idea is to build a DBG from the short reads and then find the best alignment between the long read and a path in the DBG. The sequence of the path can then be used as the corrected long read.

Zhang et al. [120] performed an evaluation of 16 different error correction methods. Based on their results, we chose FMLRC [76] as a fast and accurate hybrid error corrector for comparison. We also compare to LoRDEC [2] since our pipeline uses the same overall idea.

LoRDEC [2] builds a de Bruijn graph from the short reads, then aligns the long reads to it using a depth-first search and uses the path sequence as the corrected read. FMLRC [76] also aligns the reads to a graph, except instead of building one de Bruijn graph, it uses an FM-index which can represent all de Bruijn graphs and dynamically vary the k-mer size. FMLRC then corrects the reads in two passes, using different k-mer sizes. Our error correction pipeline is similar to LoRDEC. Figure 3.12 shows the pipeline. We first self-correct the Illumina reads using Lighter [121], then build the de Bruijn graph using BCalm2 [67], align the long reads using GraphAligner with default parameters and finally extract the path as the corrected read.

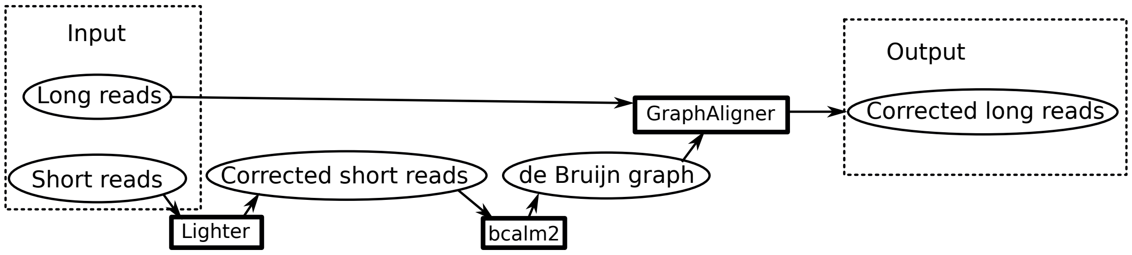


FIGURE 3.12 Overview of the error correction pipeline. The circles represent data and the rectangles programs.

Due to fluctuations and biases of Illumina coverage, some genomic areas are impossible to correct with short reads even in principle. Our pipeline has two modes: either we output the full reads, keeping uncorrected areas as is; or clipped reads, which remove the uncorrected areas and split the read into multiple corrected sub-reads, if needed. In the results, we present the full reads as “GraphAligner”, and the clipped reads as “GraphAligner-clip”. We similarly report “LoRDEC” as full reads and “LoRDEC-clip” as clipped reads. FMLRC does not offer an option to clip the reads so we report only the full reads.

To evaluate the results, we use the evaluation methodology from Zhang et al. [120]. The long reads are first corrected, and then the evaluation pipeline is run for both the raw reads and the corrected reads. The first step of the evaluation is removing reads shorter than 500 bp. Note that the reads are removed during the evaluation step, that is, they are corrected in the initial correction step and different reads may be removed in the uncorrected and corrected sets. After this, the remaining reads are aligned to the reference genome. The alignment yields several quality metrics, including number of aligned reads and base pairs, read N50, error rate and genomic coverage. Here, we report error rate as given by samtools stats instead of alignment identity. Resource consumption is measured from CPU time and peak memory use. We use the *E. coli* Illumina+PacBio dataset (E. coli, called D1-P + D1-I by Zhang et al.) and the *D. melanogaster* Illumina+ONT dataset (Fruit fly, called D3-O + D3-I by Zhang et al.) from Zhang et al. [120]. In addition, we use whole human genome PacBio Sequel[1] and Illumina[2] data from HG00733, randomly subsampled to 15x coverage for PacBio and 30x for Illumina. We use the diploid assembly from [56] as the ground truth to evaluate against for HG00733. We did not include LoRDEC in the fruit fly or HG00733 experiments as the results in [120] show that FMLRC outperforms it in both speed and accuracy. Although we use the same evaluation method, our results are slightly different. This is due to

[1]SRA accession SRX4480530
[2]SRA accessions ERR899724, ERR899725, ERR899726

Dataset	Method	# Reads	Bases (Mbp)	Aligned reads (%)	Aligned bases (%)	N50 (bp)	Genome fraction (%)	Error rate (%)	CPU time (hh:mm:ss)	Peak memory (GB)
E. coli	Original	85460	748.0	97.0	92.0	13990	100	13.1237	-	-
PacBio	LoRDEC	85316	716.5	97.9	92.9	13484	100	1.3902	10:11:28	5.0
	LoRDEC-clip	129754	654.5	99.9	99.8	8206	100	0.0881	10:11:28	5.0
	FMRLC	85260	706.5	97.7	94.8	13364	100	0.3016	4:16:43	2.6
	GraphAligner	85271	710.7	97.7	93.9	13411	100	0.5057	0:23:08	5.8
	GraphAligner-clip	91909	673.9	99.9	99.8	12146	100	0.0240	0:23:08	5.8
Fruit fly	Original	642255	4609.5	84.4	82.5	11956	98.77	16.1650	-	-
ONT	FMRLC	641956	4646.9	89.6	85.1	12087	98.62	2.3250	65:17:52	9.2
	GraphAligner	640548	4653.7	90.7	85.6	12109	98.63	1.2433	15:12:30	11.9
	GraphAligner-clip	762073	4188.3	99.3	94.7	8698	97.86	0.7087	15:12:30	11.9
HG00733	Original	2394990	48801.0	95.6	92.8	33109	95.27	13.5384	-	-
PacBio	FMRLC	2392533	48229.9	98.3	92.7	32823	95.19	7.1210	2222:13:44	234.5
	GraphAligner	2390656	48216.2	98.1	94.6	32879	94.89	3.3510	174:54:13	76.7
	GraphAligner-clip	8252956	42292.0	99.8	98.3	7973	91.91	1.3503	174:54:13	76.7

TABLE 3.4 Results of the error correction experiment. Reads shorter than 500 base pairs are discarded. The remaining reads were aligned to the reference using minimap2 [29] and the statistics were given by samtools [124] stats, except N50 which is calculated by a script from Zhang et al [120] and resource use which are measured by "/usr/bin/time -v".

two factors: First, Zhang et al. use LoRDEC version 0.8 with the default parameters, while we use version 0.9 with the parameters suggested for *E. coli* in the LoRDEC paper [2]. Second, Zhang et al. use FMLRC version 0.1.2 and construct the BWT with msBWT [122], while we use version 1.0.0 and construct the BWT with RopeBWT2 [123] as recommended by the FMLRC documentation.

Table 3.4 shows the results. The amount of aligned sequence is similar in all cases. For the PacBio data sets, the amount of corrected sequence is lower than the uncorrected input sequence, while for ONT, the amount of corrected sequence increases during correction. This is consistent with the observation that insertion errors are more common than deletions in PacBio and vice versa for ONT [125]. The number of reads is noticably higher and the N50 is lower for the clipped modes for both LoRDEC and GraphAligner, showing that most reads contain uncorrected areas and clipping the reads reduces read contiguity. In addition, the fruit fly and human experiments show that clipping the reads significantly reduces the genome fraction covered by the reads. The clipping is more pronounced in the more complex genomes, with the reads in the whole human genome dataset being on average cut into four pieces, around 4% of the genome lost due to clipping and a large reduction in read N50. We see that GraphAligner is about 30x faster and 2.7x more accurate than LoRDEC for *E. coli*. GraphAligner is over four times faster than FMLRC in all datasets. When not clipping reads, GraphAligner's error rate is slightly worse then FMLRC for *E. coli* (0.51% vs. 0.30%), but substantially better for *D. melanogaster* (1.2% vs. 2.3%) and human (3.4% vs. 7.1%). For the human genome HG00733, GraphAligner hence produces over two times better error rates while the runtime is over twelve times faster.

Our pipeline is a large improvement in runtime over the state-of-the-art. The error rates are competitive for simpler genomes and significantly better for more complex genomes. We hypothesize that the two-pass method used by FMLRC can in principle enable better correction than a single k-mer size graph, but FMLRC's performance with the larger genomes is limited by their alignment method, while GraphAligner can handle the more complex genomes. When using the clipped mode, that is, when only considering parts of the reads that have been corrected, the accuracy in the corrected areas can approach or exceed the accuracy of short reads. This emphasizes the value of this clipped mode to users. The main source of errors are in fact uncorrected areas without sufficient short read coverage.

3.10 Conclusion

In this chapter I have presented GraphAligner, a tool for aligning long reads to genome graphs. Before the publication of GraphAligner [96] there were no tools which could align long reads to de Bruijn graphs accurately and quickly. GraphAligner made alignment of long reads to genome graphs practical. In the next chapter I will describe new applications made possible by GraphAligner.

3.11 Acknowledgments

We are grateful to Rayan Chikhi for advice on building de Bruijn graphs, to Shilpa Garg for inspiring discussions and to Jana Ebler for advice on handling VCF files.

CHAPTER 4

Applications of sequence-to-graph alignment

The previous chapters presented the theoretical basis for rapid sequence-to-graph alignment, as well as GraphAligner, a tool for achieving quick and accurate sequence-to-graph alignment in practice. In this chapter I present two projects that use GraphAligner as a key component.

4.1 RNA expression quantification

The gene expression material in this chapter (Section 4.1.2, transcript quantification in Section 4.1.4) is based on work by Dilip Durai in our previous paper "AERON: Transcript quantification and gene-fusion detection using long reads" [97]. The fusion gene detection material in this chapter (Section 4.1.3, fusion detection with simulated and real data in Section 4.1.4) is reused from my work in the same paper. The other material (Sections 4.1.1, 4.1.5) is based on joint work from the same paper.

4.1.1 Introduction

Whole transcriptome sequencing is an important method in many projects, used in applications such as detection of disease specific gene expression patterns [126] and the detection of gene fusion events in cancer cells [127]. Traditionally short read technology is used for transcript sequencing. However, transcript sequences can be thousands of base pairs long, and usually undergo alternative splicing [128, 129]. While short reads can recover exome sequences and splicing junctions, they cannot recover the full transcript sequences due to short read lengths. Recently, long read sequencing technologies have been applied to RNA sequencing as well. Long read sequencing technologies can sequence molecules much longer than transcribed RNA molecules, and so can enable both resolution of transcripts over their full lengths, as well as more accurate estimations of their abundances.

In this work, we developed a pipeline for estimating transcript abundance based on long reads and splice graphs. We also detect gene fusions based on aligning long reads to a graph, a novel method that has not been demonstrated so far.

4.1.2 Expression quantification

Figure 4.1 shows an overview of our transcript quantification pipeline. The first step of AERON is to build an index. The input of the index construction is a reference genome, a set of reference transcript sequences and their genes, and the positions of the exons of the reference transcripts. The index consists of a *splicing graph* for each gene and the alignments of the transcripts to the graphs. We build a splicing graph called a *gene-exon* graph from each gene. The gene-exon graph contains all the exons of a gene split at every splice site. The split exons are then connected based on their position in the genome. Each split exon is connected to all split exons after it. The gene-exon graph represents all potential splicings of a gene, not just those represented in the reference annotation.

More formally[1], given a genomic string g and a list of n exon start and end positions $(s_1, e_1), ..., (s_n, e_n)$ for one gene, define a list of *border sites* as $B = (s_1, e_1, ..., s_n, e_n)$. Define a list of border sites sorted by genomic position $B' = (b_1, ..., b_{2n})$. Given the list of border sites sorted by genomic position, every adjacent pair of border sites covered by an exon results in one vertex in the gene-exon graph, that is, the vertex set is $V = \{v_i : 1 \leq i < 2n \wedge \exists_j (s_j \leq b_i < b_{i+1} \leq e_j)\}$ Each vertex v_i is labeled by the genomic substring from b_i to b_{i+1}: $\sigma(v_i \in V) = g[b_i, b_{i+1})$. The edge set of the gene-exon graph is $E = \{(v_i, v_j) : v_i \in V \wedge v_j \in V \wedge i < j\}$. Figure 4.2 shows the construction of a gene-exon graph for one gene.

Once the gene-exon graphs have been built for all genes, the reference transcripts are aligned to them. GraphAligner is used for aligning the transcripts. Taken together the collection of all gene-exon graphs and the alignments of the transcripts is called the index.

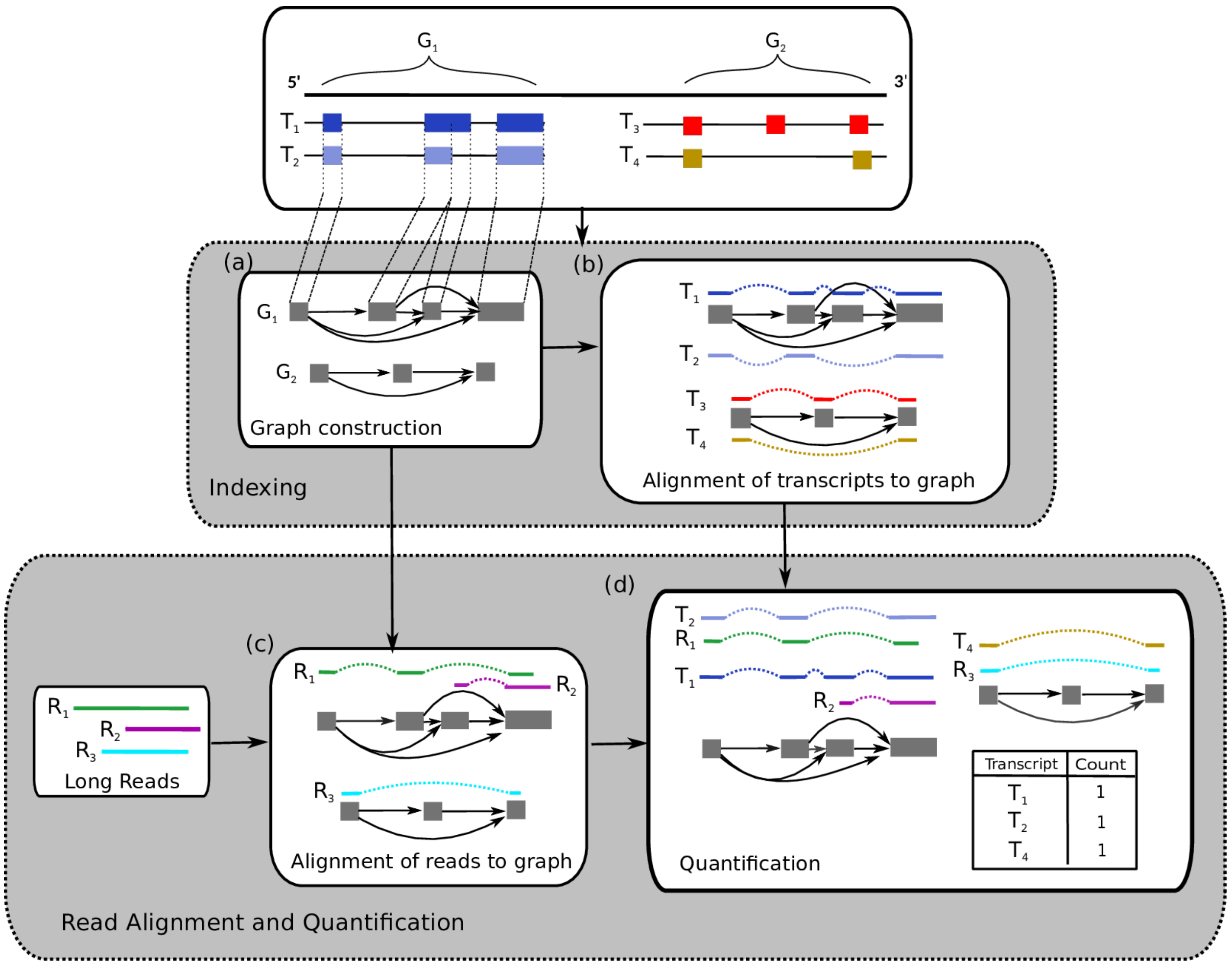
Given the index and the reads, the transcript expression can be estimated. The reads are aligned to the collection of gene-exon graphs. Alignments are discarded if the *E-value* [130] of the alignment is more than 1. The E-value estimates the number of spurious alignments, and is calculated with the formula

$$E = Kmne^{-\lambda S} \tag{4.1}$$

where E is the expected number of spurious hits, K and λ are parameters that depend on the scoring scheme, S is the alignment score and m is the database size in base pairs and n is the query size in base pairs. The original formula by Karlin and Altschul is defined with a database of linear sequences instead of graphs. We use the number of base pairs in the graph as the database size. The K and λ parameters were chosen to correspond to a scoring scheme with match score +1 and mismatch cost -2.

At this point, the base pair sequences of the reads are discarded and only the alignment paths

[1]The definition used here is different but equivalent to the definition used in the publication [97]



Transcript	Count
T$_1$	1
T$_2$	1
T$_4$	1

FIGURE 4.1 Workflow of the transcript quantification step of AERON. Top: The input of the indexing step is the reference genome and a list of gene annotations. The gene annotations contain the positions of the exons along the reference. Middle: The reference is indexed by building *gene-exon graphs*, and aligning the transcripts to the gene-exon graphs. Bottom: To estimate the transcript expression, long reads are first aligned to the gene-exon graphs. Then, the read alignment paths are compared with the transcript alignment paths. Each read is assigned to one transcript and the reads per transcript are counted. Figure by Dilip Durai [97].

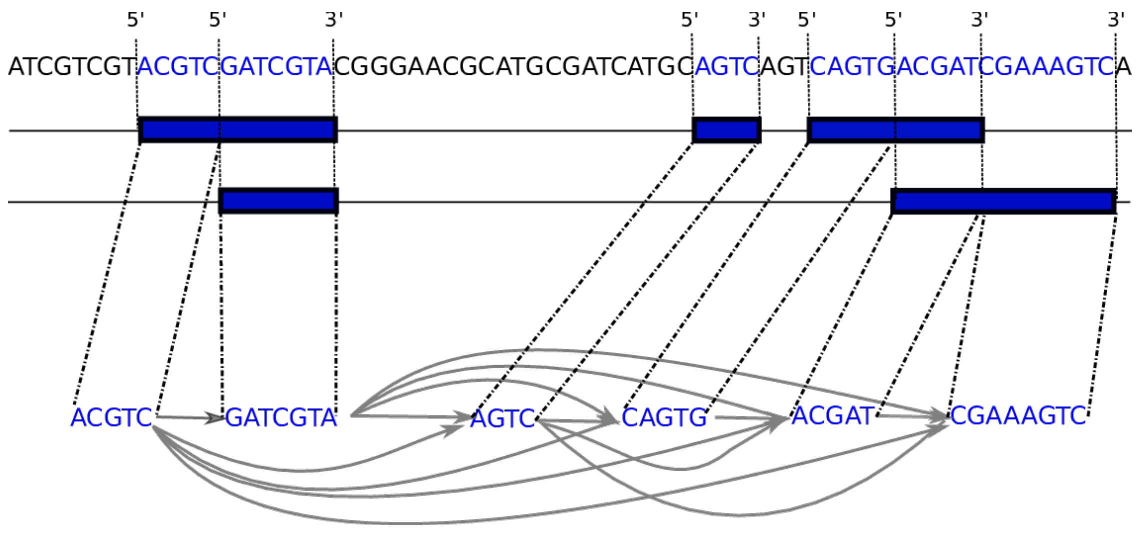

5'
5'
3'
5'
3'
5'
5'
3'
3'
ATCGTCGTACGTCGATCGTACGGGAACGCATGCGATCATGCAGTCAGTCAGTGACGATCGAAAGTCA
ACGTC
GATCGTA
AGTC
CAGTG
ACGAT
CGAAAGTC

FIGURE 4.2 Construction of a gene-exon graph from the reference and a set of exons. Top: the reference genome. Base pairs which are covered by an exon are marked in blue and uncovered in black. Middle: blue boxes represent exons. The exons can overlap with alternate splicing. Bottom: the resulting gene-exon graph. Each splice boundary splits an exon into a node. The nodes are then connected according to their order in the reference genome, with every node connected to all nodes after it. Figure by Dilip Durai [97]

are kept. Given a read which aligns to the path $(v_1, v_2, ..., v_n)$, the read is transformed into the string $v_1 v_2 ... v_n$. The reference transcript alignments are similarly transformed into strings from their alignment paths. The transformed reads and transcripts are then aligned to each other with semi-global Needleman-Wunsch alignment, aligning the entire read to a substring of the transcript. The costs of mismatches and edits are based on the lengths of the nodes. That is, for a node v_1 of length n, inserting or deleting node v_1 has a cost of n and substituting v_1 for v_2 of length m has a cost of $\max(m, n)$. The alignment between the read and the transcript is then used to compute an *alignment score* between the read and the transcript. The alignment score is the number of base pairs in the matches of the alignment divided by the read length. The alignment score essentially describes whether the read is a subsequence of the transcript; a score of 1 means that the read is entirely contained within the transcript, and a score of 0.9 means that 10% of the read sequence does not match the transcript. Overlaps with an alignment score less than 0.2, meaning less than 20% of the read overlaps with the transcript, are discarded.

A read can have the same alignment score with multiple transcripts. In this case, the distance of the read from the 3' end of the transcript is used to decide between them. Each read is then assigned to the one transcript with the highest alignment score and shortest distance from the 3' end. In case of ties the read is arbitrarily assigned to one of the transcripts.

The read counts per transcript are then transformed to a normalized *transcripts per million* (TPM) metric. The TPM metric is a measure of transcript expression normalized to coverage. A TPM value of x for a transcript means that with a million transcript molecules, on average x of them come from the transcript. Given the count of reads per transcript $c_1, c_2, ..., c_n$, the TPM for transcript i is calculated as $TPM(i) = 1000000\frac{c_i}{\Sigma_x c_x}$.

Once the expression per transcript has been computed, the expression per gene can also be computed. The expression per gene is simply the sum of the expressions of its transcripts.

4.1.3 Fusion gene detection

The second part of the AERON pipeline is detecting fusion genes using long reads. There are three main steps in the fusion detection pipeline: First, partial alignments of the reads to the gene-exon graphs provide a list of *tentative fusions*. Next, the reads are re-aligned to *fusion graphs* derived from the tentative fusions. Finally, the alignments to the fusion graphs are compared with alignments to gene-exon graphs to provide a *fusion score* for each read. Each of the steps produces a list of fusion event candidates, with the later steps filtering out candidates from the previous steps. Figure 4.3 shows an overview of the fusion detection pipeline, which we explain in the following.

Partial alignments

The reads are first aligned to the gene-exon graphs. This proceeds the same way as in the quantification pipeline, except that secondary alignments, where a read may be aligned to multiple places with different alignment qualities, are kept and the same part of a read may be mapped to multiple gene-exon graphs.

Tentative fusions

The partial alignments are used to create a list of *tentative fusions*. Whenever a read has a pair of partial alignments whose endpoints in the read are within 20 base pairs to each other, and where the two parts are aligned to different genes, the read supports a tentative fusion between the two genes. Each read may support multiple tentative fusions.

Fusion graphs

Each tentative fusion induces a *fusion graph*. The fusion graph combines the gene-exon graphs of the two participating genes. An extra crossover node is added to connect the two gene-exon graph. Each base pair in the first gene-exon graph is connected to the crossover node, and the crossover node is connected to each base pair in the second gene-exon graph. This way, the alignment may cross from any point in the first gene-exon graph to any point in the second gene-exon graph.

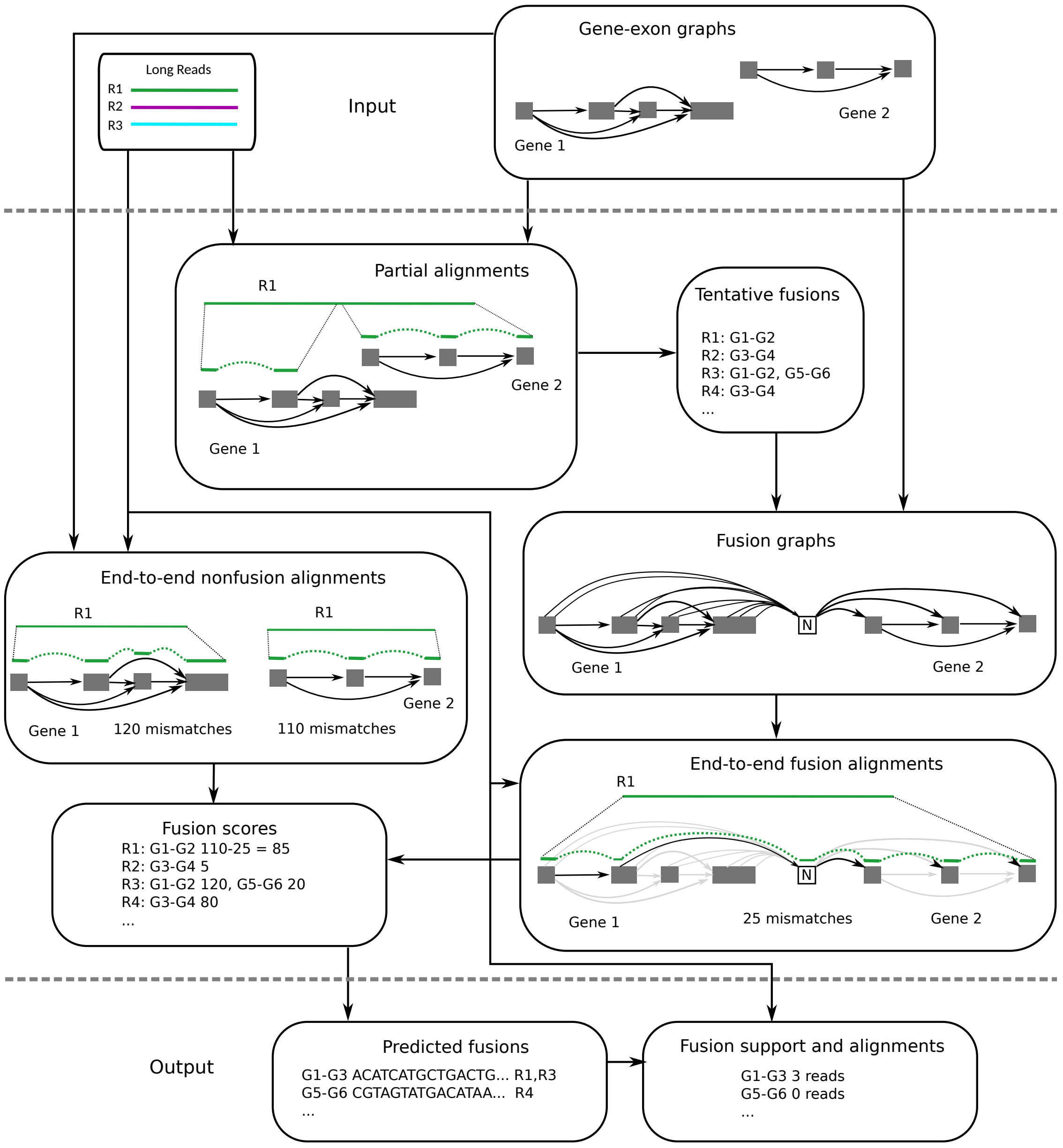


FIGURE 4.3 Workflow of the fusion detection step of AERON. Partial alignment: reads are aligned to the gene-exon graphs. All secondary alignments are kept and the read may have alignments to different genes. Tentative fusions: whenever a read has a pair of alignments that end within 20 bp of each other in the read, the read votes for a fusion between the two genes. A read may vote for multiple tentative fusions. Fusion graphs: each tentative fusion induces a fusion graph, where the two genes are connected with a crossover node (N). End-to-end fusion alignments: the reads are aligned to the fusion graphs. Global alignment is used to align the read from start to end. End-to-end nonfusion alignments: the reads are aligned to the individual gene-exon graphs globally. Fusion score: the score difference between the fusion alignment and the nonfusion alignment defines a fusion score. Predicted fusions: the alignments are filtered based on the fusion score. The graph sequence along the alignment is taken as the predicted fusion transcript. Fusion support and alignments: all reads are aligned to the reference transcripts and the predicted fusion transcripts with Minimap2. A read supports a fusion if its primary alignment covers the fusion breakpoint with at least 150 base pairs on both sides.

End-to-end fusion alignments

The reads are aligned to their fusion graphs. However, here the alignment must span the entire read from start to end. Clipped read ends are considered indels and contribute to the number of mismatches in the alignment. The point of this step is to quantify “how well” the fusion graph can explain the read. A read which comes from a fusion event between the two genes must necessarily have a low edit distance to some path in the graph, and therefore the edit distance of the read’s alignment to the fusion graph will be low. However, a read which does not come from a fusion event between the two genes does not necessarily have a high edit distance to the fusion graph. For example, a read which was transcribed from one of the genes will have a low edit distance to the fusion graph as well. To filter out these cases, the reads must be aligned to the nonfusion graphs as well.

End-to-end nonfusion alignments

Each read supported a list of tentative fusions previously. We extract the list of genes involved in those fusions for each read. In addition to this, we extract each pair of tentative fusions which include those genes regardless of which read supports the fusion, and say that these genes are relevant for the read. That is, if read R_1 supports a fusion between genes G_1 and G_2 and nothing else, but an another read supports a fusion between G_2 and G_3, then all of G_1, G_2 and G_3 are relevant for R_1. The reads are then aligned to all of their relevant gene-exon graphs. Again the alignments must span the entire read from start to end. A read which comes from a non-fused gene will have a low edit distance to some gene-exon graph. However, a read that comes from a fusion event will have a high edit distance when aligned to just one of its gene-exon graphs.

Fusion scores

Once the reads have been aligned end-to-end to both the fusion graphs and the gene-exon graphs, the alignment scores are compared to calculate a *fusion score*. Given the lowest alignment edit distance to a fusion graph C_f and the lowest alignment edit distance to a gene-exon graph C_n, the fusion score of a read to the fusion graph is defined as $C_n - C_f$. This essentially describes how much better the read aligns to a fusion than any individual gene; a fusion score of 0 means that the read aligns to a fusion graph just as well as to a non-fusion graph, and higher fusion scores mean that the read aligns better to the fusion graph than any non-fusion graph. Note that the alignment edit distance to a fusion graph cannot be worse than the edit distance to a gene-exon graph, since the fusion graphs include the gene-exon graphs as subgraphs. The fusion graph alignment with the lowest edit distance is kept and the others are discarded. At this point each read can only support one fusion, which removes a large number of false positives as seen in the simulated fusion experiment in Section 4.1.4.

Predicted fusions

The reads are filtered based on the error rate of the alignment to the fusion graph and the fusion score. Reads whose fusion score is below a user-given threshold (default 200) are discarded. Reads whose alignment error rate to the fusion graph is above 20% are also discarded. The paths of the fusion alignments are taken as the predicted fusion transcripts. When multiple reads align to the same fusion graph and cross over at the same exon, they are considered the same fusion event and one of them is arbitrarily selected as the fusion transcript. If multiple reads align to the same fusion graph but cross over at different exons, they are considered separate events. The output of this step is a list of predicted fusion transcripts.

Fusion support and alignments

Finally, all reads are aligned to the predicted fusion transcripts and the reference transcriptome using Minimap2 [29]. A read is then considered to support a fusion if its primary alignment crosses the fusion breakpoint with at least 150 base pairs on both sides. This recovers some reads which were missed by the earlier steps and removes some spurious fusions. The output of this step is a BAM file containing the alignments of the reads to the transcriptome and predicted fusion transcripts, and the number of reads that support each predicted fusion transcript.

4.1.4 Results

Transcript quantification

The transcript quantification experiment was performed by Dilip Durai. Here I briefly summarize the results.

We ran the transcript abundance quantification pipeline on two Oxford Nanopore Technologies transcript sequencing datasets. The datasets were from the individual NA12878, and the cancer cell line K562. As a comparison, we used minimap2 [29] to align the reads to the transcripts and used a similar method to quantify expression based on the alignments. We compared the expression estimations of AERON and minimap2 to the estimations of Salmon [131], a transcript expression quantification tool using short reads.

To evaluate the results, we compared the Salmon estimation to the AERON or minimap2 estimation using two metrics. First, we used Spearman correlation. Second, we used Mean Absolute Relative Difference (MARD). Given two lists of numbers x and y of length M, the MARD is defined as

$$MARD(x,y) = \frac{1}{M}\sum_{i=1}^{M}\begin{cases}0, & \text{if } x_i = y_i = 0\\ \frac{|x_i - y_i|}{x_i + y_i}, & \text{otherwise}\end{cases},$$

Unlike in correlation, a lower MARD means lower error, and a higher MARD means higher error.

Figure 4.4 shows the comparison on gene-level expression on K562 and NA12878, and Table 4.1 shows the correlation and MARD. AERON is more accurate than minimap2 in both datasets. AERON aligned most of the reads in the NA12878 dataset, but only about half of the K562 reads. This is partially due to poorer quality reads in the K562 dataset [97], and potentially due to novel events in the K562 cancer cell line which are not properly reflected by the refererence transcriptome.

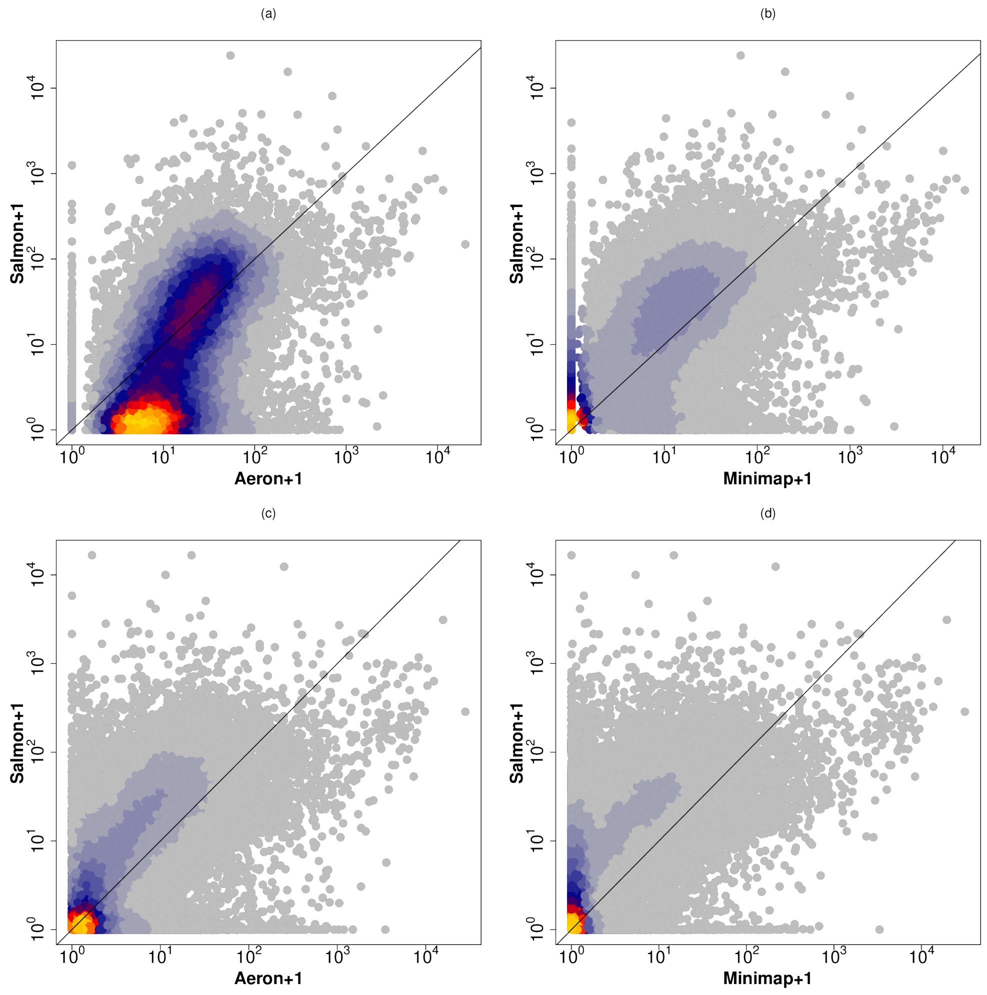


FIGURE 4.4 Results of the transcript quantification experiment with AERON. Each plot compares the quantification of either AERON or minimap2 against Salmon. Each plot shows a heatmap where each gene is represented by one data point. Top row: estimates with the K562 dataset. Bottom row: estimates with the NA12878 dataset. The axes use logarithmic scales. To show data points with zeros, each value has one added to it. Figure by Dilip Durai [97].

Dataset	AERON			Minimap2		
	#reads mapped	Correlation	MARD	#reads mapped	Correlation	MARD
K562 (2.7M)	2,167,286	**0.833**	**0.350**	1,074,411	0.704	0.428
NA12878 (25M)	23,902,112	**0.822**	**0.349**	23,211,716	0.778	0.37

TABLE 4.1 Spearman correlation and MARD between Transcripts Per Million (TPM) at gene level obtained from AERON/Minimap2 using Oxford Nanopore Sequencing (ONT) data and TPM at gene level obtained from Salmon using Illumina data. The size of the dataset is depicted in brackets next to the name. Table by Dilip Durai [97].

Fusion detection on simulated data

We first assessed the performance of our fusion detection approach in a simulation study. We generated fusion events of different "lengths", where the length refers to the amount of sequence from both genes. For example a fusion event with length 200 bp contains 200 base pairs from both genes and has a total length of 400 bp. Events were simulated in 9 length groups, from 100-200 bp, 200-300 bp, and so on until 900-1000 bp. For each of these length ranges, 50 fusions were generated, where a pair of transcripts was selected randomly for each fusion. Then, a random substring of each transcript corresponding to the length of the fusion was selected, and the substrings were concatenated to build the fusion transcript. The reads were simulated at 10$\times$ coverage from all simulated fusion and reference transcripts. The fusion detection pipeline was then ran on the simulated reads.

The left part of Fig. 4.5 shows the precision-recall curve at varying fusion score cutoffs for different fusion sizes. We see that 100-400 base pair fusions (bottom) are hard to detect with any fusion score cutoff, and the recall saturates at 15%. The high error rate and short length of the reads stops them from being aligned in the tentative fusion phase, which prevents the pipeline from detecting the fusions. However, 400-700 base pair fusions (middle) are detected, and the recall reaches up to 87%. For the longer fusions of 700-1000 base pairs (top), recall reaches up to 95% and precision around 80%. Based on the curves, we chose 200 as the default fusion score cutoff, as that achieves a precision of 78% and recall of 90% for large fusions.

The right part of Figure 4.5 shows the number of detected fusions (true positives) as a function of fusion length at different phases of the fusion detection pipeline. These experiments also show that our three-step approach progressively removes false positive fusion events: while there are 28,696 false positive predictions in the tentative step, this number reduces to 49 in the graph step and further down to 20 in the final step. We see that shorter fusions are not detected even in the tentative fusion phase; this is most likely due to the high error rate preventing short alignments from being found. Once the fusion size grows above 400 bp, the set of tentative fusions contain most of the fusion events. Importantly, the fusion graph approach removes only a small fraction of true fusions, while removing almost all false positives from the tentative fusions. The difference

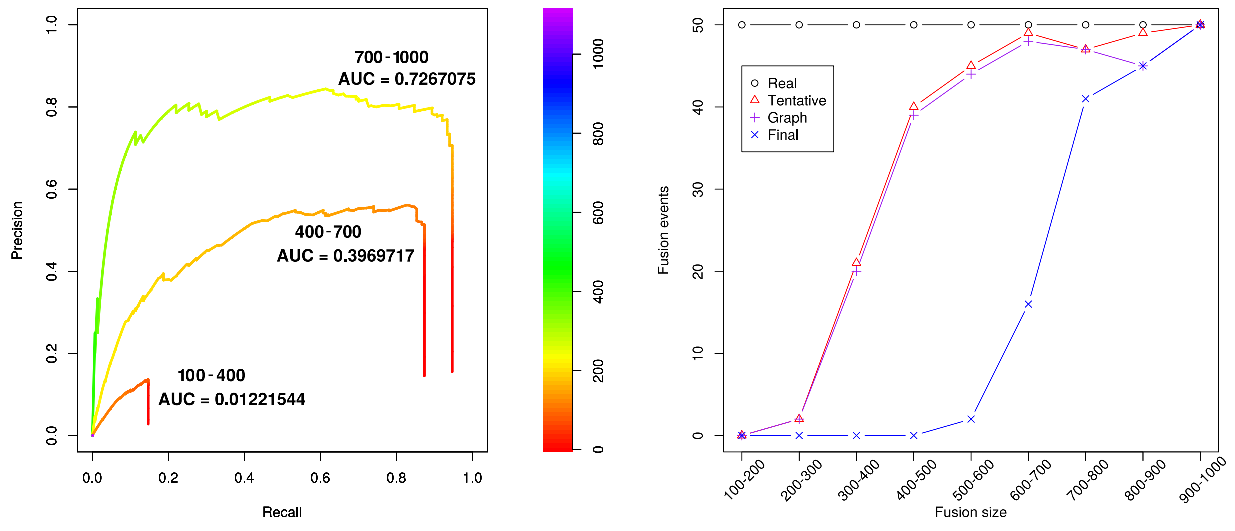


FIGURE 4.5 Left: Precision-recall curves for fusion event detection with simulated data for fusions of 100-400 base pairs (bottom), 400-700 base pairs (middle) and 700-1000 base pairs (top). Both precision and recall improve for longer fusions. The parameter varied is the fusion score cutoff. Right: number of detected true fusion events per fusion size with simulated data. The curves show the number of simulated fusions (Real) and fusions detected at different parts of the pipeline: *tentative fusions* (Tentative), after fusion graph alignment (Graph), and after filtering for fusion score (Final). The number of total false positives is 28696 for Tentative, 49 for Graph and 20 for Final.

in the “Graph” and “Final” curves shows that the fusion score cutoff further removes more false positives, but at the cost of removing shorter true positives as well.

Fusion detection on real data

We ran the fusion detection pipeline on two datasets: one from the human individual NA12878 and one from the cancer cell line K562. The NA12878 functions as a control, as we do not expect to see any fusions in that dataset. K562, on the other hand, is a highly rearranged cancer cell line [132] with the known BCR-ABL1 fusion.

We ran the pipeline and took all predictions which were supported by at least two reads, resulting in 25 events for K562 and 24 for NA12878. We then further manually curated the predictions in three steps. First, we removed all fusions involving a mitochondrial gene and removing fusions between a gene and its own pseudogene, leaving 16 events for K562 and 14 for NA12878. Then we used IGV [133] to visually inspect the alignments of the reads to the predicted fusion transcripts generated by the pipeline, and discarded events where the alignments to one of the genes seemed noticeably worse than the other or the read coverage over the breakpoint was noticeably less than for either of the genes. Figure 4.6 shows an example of an event that was rejected based on the IGV visualization. After IGV visualization, 15 events were left for K562 and 9 for NA12878.

Since some transcripts can have similar sequences, the high error rate of the reads can cause a false positive fusion prediction between the two transcripts due to sequence similarity. In this case, the predicted fusion transcript should be similar to a reference transcript, because it reflects a reference-guided consensus between the long reads. Hence, the average error rate of the fusion transcript is much lower than the input reads and therefore easier to align with traditional methods. In order to detect transcripts not annotated in the Ensembl release, we used the BLAST webserver to align the predicted fusion transcripts to the human reference genome (GRCh38.p12) and transcriptome (NCBI Homo sapiens Annotation Release 109). We discarded events that mapped to an existing transcript including the fusion breakpoint and retained 8 events for K562 and 2 events for NA12878.

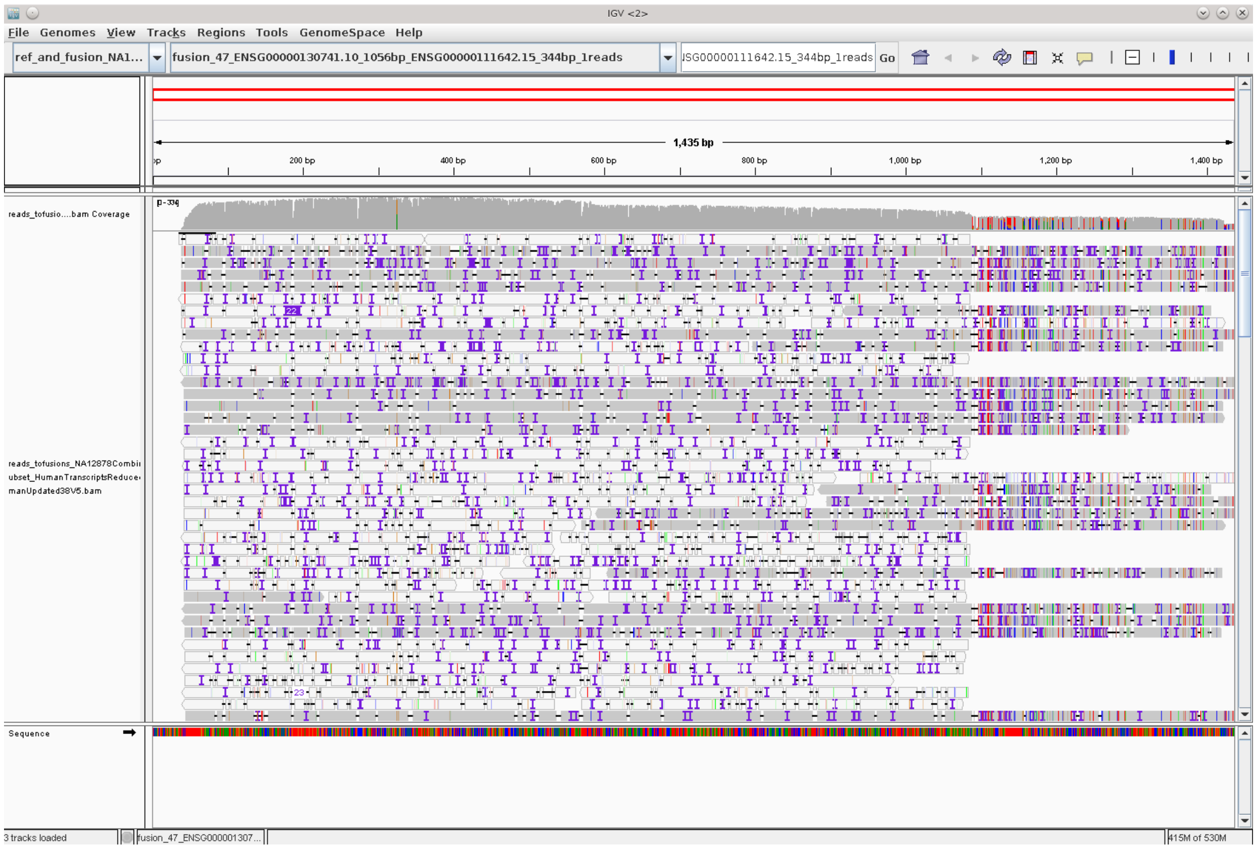

IGV <2>
File Genomes View Tracks Regions Tools GenomeSpace Help
ref_and_fusion_NA1...
fusion_47_ENSG00000130741.10_1056bp_ENSG00000111642.15_344bp_1reads
ISG00000111642.15_344bp_1reads
Go
1,435 bp
200 bp
400 bp
600 bp
800 bp
1,000 bp
1,200 bp
1,400 bp
reads_tofusio....bam Coverage
reads_tofusions_NA12878Combi
ubset_HumanTranscriptsReduce
manUpdated38V5.bam
Sequence
3 tracks loaded
fusion_47_ENSG000001307...
415M of 530M

FIGURE 4.6 A predicted fusion gene which was rejected by manual curation. The alignments to the right part of the fusion are clearly worse than those to the left part, showing that the alignments to the predicted fusion are spurious.

The two predicted fusion events for NA12878 are shown in Table 4.2. Figures 4.8 and 4.9 show corresponding IGV screenshots of the transcripts and their reads. The fusions appear to be well supported by a few reads. However, we believe nonetheless that these are false positives. The fusions might be caused by chimeric reads.

Table 4.3 shows the eight predicted fusion events for K562 including the well-known BCR-ABL1 fusion event (Fig. 4.7). Four of the eight predicted fusion events have been reported in literature before. For the BCR-ABL1 fusion, the TEN1-CDK3 read-through and BMS1P4-AGAP5

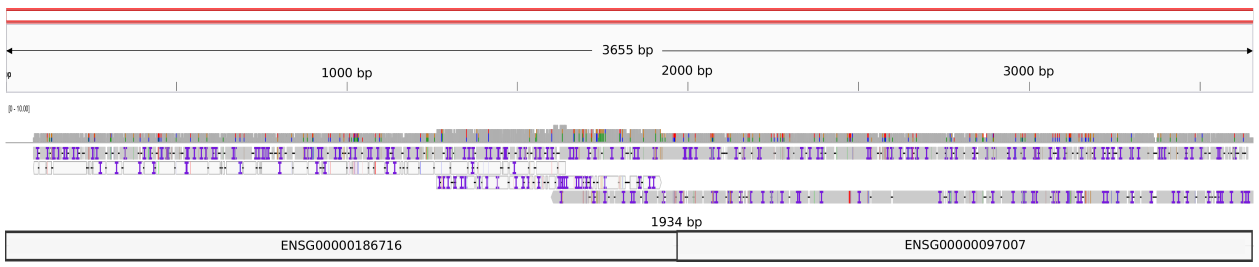


FIGURE 4.7 The coverage plot and the alignment of reads against the 3655bps long BCR-ABL1 fusion transcript. The fusion breakpoint was found to be at the position 1934. The image was generated using Integrated Genomics Viewer(IGV).

Ensembl ID	Gene	Chr.	Ensembl ID	Gene	Chr.	Support
ENSG00000223361	FTH1P10	chr5	ENSG00000162734	PEA15	chr1	3 reads
ENSG00000172493	AFF1	chr4	ENSG00000162244	RPL29	chr3	2 reads

TABLE 4.2 Predicted fusion events for NA12878. The predicted fusion transcripts do not have BLAST hits that cover the fusion breakpoint. The first 6 columns describe the two genes involved in the fusion. The column "Support" counts the number of reads whose primary alignment covers the fusion breakpoint and 150bp from both sides of it.

read-through events, the AERON predictions mapped to the existing annotations. The NOS3-PRKN predicted fusion mapped to a transcript variant of the NOS3 gene, while the ARPC4-TTLL3 predicted fusion mapped to a transcript variant of ARPC4. The PRIM1-NACA predicted fusion occurred with fusions across two different breakpoints, which the pipeline considers separate events. The HBG2-HBG1 predicted fusion has a very high read support. Figure 4.10 shows a dot plot of the alignment of the predicted transcript to the reference, which is consistent with an inverted duplication occurring in the region.

4.1.5 Discussion

AERON uses a novel sequence graph based method for quantifying RNA expression with long reads. The RNA expression quantification is slightly more accurate than the linear alignment based method using minimap2. AERON also uses a novel method for detecting gene fusion events with long reads. Methods for detecting gene fusion events with long reads have not been presented before. The experiment on the K562 cancer cell line detected the known BCR-ABL1 gene fusion, as well as a few other known events. The gene fusion pipeline recovered some readthrough events as well, which were classified as gene fusions since the pipeline does not know that the genes are adjacent to each other in the genome. However, some small amount of false positives remain. The simulated data experiment produced 20 false positive calls even after the fusion score filtering. We recommend manually inspecting the predictions to filter out more false positives. In the K562

Ensembl ID	Gene	Chr.	Ensembl ID	Gene	Chr.	Support	Known
ENSG00000196565	HBG2	chr11	ENSG00000213934	HBG1	chr11	89 reads	[134]
ENSG00000186716	BCR	chr22	ENSG00000097007	ABL	chr9	2 reads	[135]
ENSG00000257949	TEN1	chr17	ENSG00000250506	CDK3	chr17	2 reads	[136]
ENSG00000204177	BMS1P1	chr10	ENSG00000188234	AGAP4	chr10	2 reads	[137]
ENSG00000241553	ARPC4	chr3	ENSG00000214021	TTLL3	chr3	4 reads	–
ENSG00000198056	PRIM1	chr12	ENSG00000196531	NACA	chr12	3 reads	–
ENSG00000198056	PRIM1	chr12	ENSG00000196531	NACA	chr12	3 reads	–
ENSG00000164867	NOS3	chr7	ENSG00000185345	PRKN	chr6	2 reads	–

TABLE 4.3 Predicted fusion events for K562. The TEN1-CDK3 and BMS1P1-AGAP4 were reported earlier as read-through events. The first 6 columns describe the two genes involved in the fusion. The column "Support" counts the number of reads whose primary alignment covers the fusion breakpoint and 150bp from both sides of it.

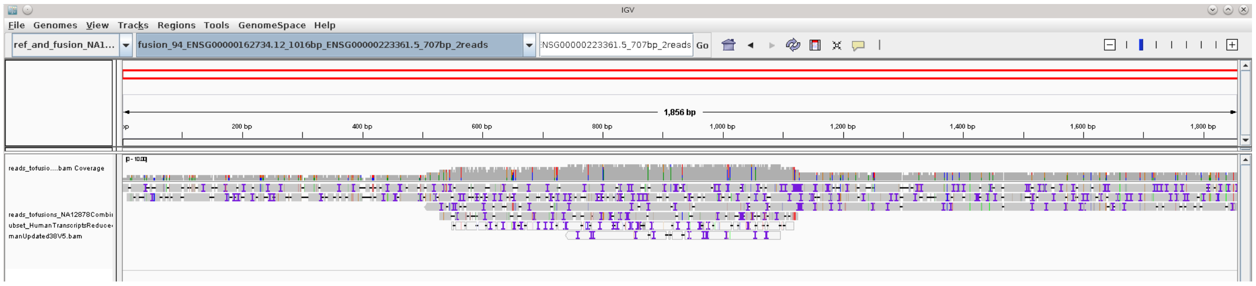

IGV
File Genomes View Tracks Regions Tools GenomeSpace Help
ref_and_fusion_NA1...
fusion_94_ENSG00000162734.12_1016bp_ENSG00000223361.5_707bp_2reads
NSG00000223361.5_707bp_2reads
Go
1,856 bp
200 bp
400 bp
600 bp
800 bp
1,000 bp
1,200 bp
1,400 bp
1,600 bp
1,800 bp
reads_tofusio....bam Coverage
reads_tofusions_NA12878Combi
ubset_HumanTranscriptsReduce
manUpdated38V5.bam

FIGURE 4.8 An IGV screenshot of the predicted FTH1P10-PEA15 fusion transcript from NA12878. The fusion breakpoint is around 1130 bp.

and NA12878 experiments, we curated the predictions by visualizing them with IGV [133] to remove cases where reads align poorly to one side of the fusion, and by aligning the predicted fusion transcripts to the reference genome and transcriptome using BLAST [37] to remove false positives caused by sequence similarity between transcripts. The gene fusion experiment on the NA12878 data resulted in 2 gene fusion events, which are likely to be false positives.

4.2 Genome assembly

In this section I describe an unpublished hybrid genome assembly pipeline using de Bruijn graphs and graph alignment.

4.2.1 Introduction

Genome assembly is one of the most fundamental problems in bioinformatics. The goal of genome assembly is to infer the genome of an organism based on sequenced reads. Many bioinfor-

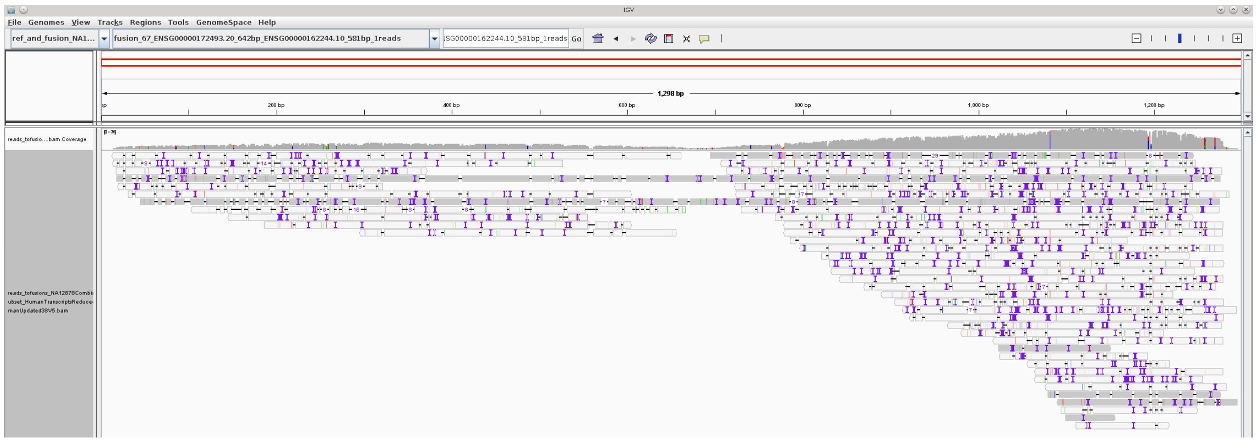
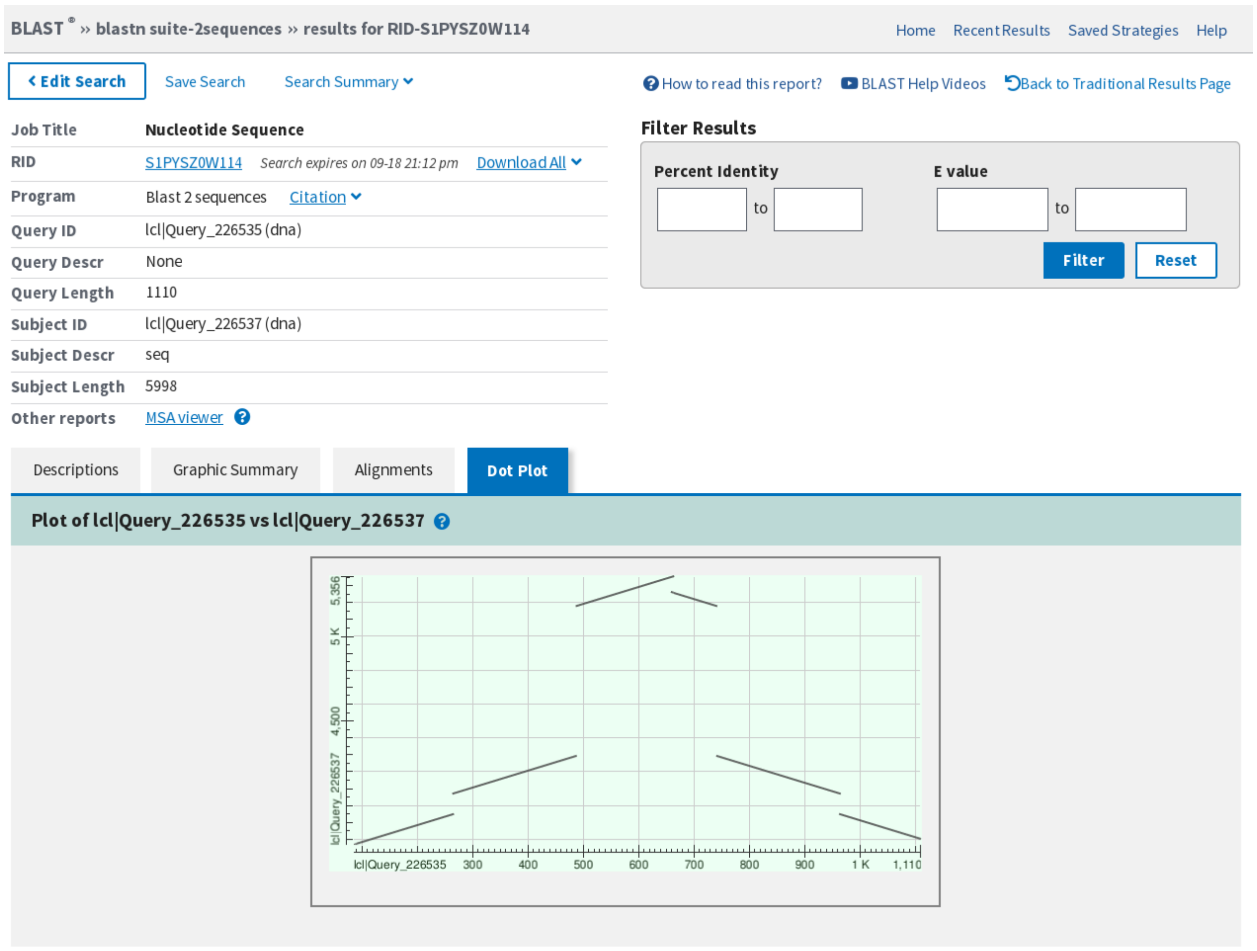

IGV
File Genomes View Tracks Regions Tools GenomeSpace Help
ref_and_fusion_NA1...
fusion_67_ENSG00000172493.20_642bp_ENSG00000162244.10_581bp_1reads
Go
1,298 bp
200 bp
400 bp
600 bp
800 bp
1,000 bp
1,200 bp
reads_tofusio....bam Coverage

FIGURE 4.9 An IGV screenshot of the predicted AFF1-RPL29 fusion transcript from NA12878. The fusion breakpoint is around 700 bp.

matics pipelines require a reference genome, and so assembling an organism's genome is generally the first step in analysing a species. Genome assembly can also be useful even when a reference genome is already available. Assembly based methods can shed light on genomic diversity on humans, and methods assembling reads that do not map to the reference genome have found significant amounts of novel sequence [56], up to hundreds of millions of base pairs [57] or about 10% of the size of the human genome. Currently, some projects aim to sequence and assemble tens of thousands of species [138, 139]. The goal of genome assembly has traditionally been a single *haploid* reference genome, but recent methods [140–143] have aimed at *haplotype-resolved* or *phased* assemblies, where the sequences of the homologous chromosomes are separated.

The available data types have affected the development of assembly algorithms. In the 2000's, short read assembly based methods using whole genome shotgun sequencing were common. The main problem with short reads is their short length, around a hundred base, which prevents repeats longer than the read length from being solved. On the other hand, short reads are highly accurate with error rates less than 1%. Paired end reads mitigated the problem somewhat and enabled more complete assemblies [68]. Within the last several years, long read sequencing technologies have become more prominent, and most recent assemblers use long reads. Long reads, such as PacBio CLR and Oxford Nanopore Technology (ONT) reads, can assemble the genome more contiguously due to spanning repeats which cannot be resolved by short reads. Earlier long read sequencing technologies achieved read lengths of few thousands of base pairs while current ones routinely reach tens of thousands of base pairs. However, CLR and ONT reads suffer from high error rates, with earlier technologies having error rates between 10%-30% and modern technologies between 5%-15%. Recently, PacBio HiFi reads have achieved long read lengths of tens of thousands of base pairs with low error rates of less than 1% [115], and recent assemblers have exploited these properties [144–146]. Read lengths have also increased, with recent ultralong ONT reads reaching lengths of hundreds of thousands of bases, and up to millions of base pairs, which has been instrumental



BLAST ® » blastn suite-2sequences » results for RID-S1PYSZ0W114

Home Recent Results Saved Strategies Help

‹ Edit Search Save Search Search Summary ⌄

How to read this report? BLAST Help Videos Back to Traditional Results Page

Job Title	Nucleotide Sequence
RID	S1PYSZ0W114 Search expires on 09-18 21:12 pm Download All ⌄
Program	Blast 2 sequences Citation ⌄
Query ID	lcl|Query_226535 (dna)
Query Descr	None
Query Length	1110
Subject ID	lcl|Query_226537 (dna)
Subject Descr	seq
Subject Length	5998
Other reports	MSA viewer

Filter Results

Percent Identity ___ to ___ E value ___ to ___

Filter Reset

Descriptions Graphic Summary Alignments Dot Plot

Plot of lcl|Query_226535 vs lcl|Query_226537

FIGURE 4.10 A BLAST screenshot showing the dot plot of the alignment between the predicted HBG2-HBG1 fusion event in the K562 data (x-axis) and chromosome 11 around 5,499,607-5,505,605 (y-axis) which contains the end of the HBG2 gene. The alignment is consistent with an inverted duplication of the region.

in resolving some of the complicated centromeric repeats [147]. Methods which combine different data types, such as short reads and long reads, are called *hybrid* assembly methods.

Although the "ordinary" sequencing technologies of short reads and long reads are used for assembling the genome into contigs, other "exotic" technologies can be used for scaffolding the contigs into chromosome scale assemblies. These technologies provide long range connectivity information, potentially over entire chromosomes, but do not deliver it in the form of a single read sequence spanning the entire molecule. Optical maps [148] can provide long range connectivity information over hundreds of thousands [149] of base pairs, but the information is limited only to the positions of certain sequence motifs. Linked reads [150] produce groups of short reads which are sequenced from the same molecule of DNA, and can span hundreds of thousands of base pairs with sparse coverage. Chromosome conformation capture [151] sequences pairs of short reads, which are more likely to be sequenced from positions close to each other in the three-dimensional structure of DNA, and can be used for assigning contigs into chromosomes with orientations and approximate positions [152]. Strand-Seq [153, 154] provides short reads sequenced from each of the homologous copies of a chromosome such that the reads from the same molecule are sequenced from the same strand in the same direction, which enables phasing in a genome wide manner [155, 156] and assembling haplotype phased genomes by clustering contigs and reads into chromosomes and haplotypes [143, 157].

Various metrics exist for assembly quality measurement. The base pair level accuracy of the genome can be measure in QV (quality value) scores similar to Phred quality scores [158], describing the average error rate. Structural correctness of assemblies can be measured by *misassemblies* or *misjoins*, which describe positions in the assembly where two separate areas of the genome are joined next to each other. The contiguity can be measured by measures like *N50*, which describes the length of the shortest contig (or scaffold) such that contigs (scaffolds) longer than the cutoff contain at least 50% of the sequence. N50 depends on the assembly size, which can be misleading if the assembly size is different from the real genome size. This is usually fixed by considering the *NG50*, where the length is compared to the genome size instead of assembly size. Ideally the *NGA50* measure is used, which aligns the contigs to a reference genome and uses the lengths of the alignments instead of lengths of contigs, which penalizes misassemblies. However, NGA50 can only be used if a high quality reference already exists, so while it is useful for benchmarking novel methods with existing data, it is not usable for assessing the quality of an assembly of a novel genome. For haplotype phased assemblies, a similar metric of *phased NG50* can be used for phasing contiguity. In addition, phasing correctness can be measured using the *switch error* rate which describes the average frequency of haplotype switches. Assembly correctness can also be measured by detecting conserved genes from the assembly [159]. Automated tools exist for evaluation of many metrics [160].

Different methods and algorithms exist for graph-based genome assembly. De Bruijn graph based methods [66] are used by several short read and hybrid assemblers [68, 83, 161]. These methods split the reads into small k-length substrings called *k-mers*, which can be efficiently represented by a graph that connects k-mers with an exact overlap of $k-1$ characters. Overlap graph [107] or string graph based methods are common for long read assemblers [47, 103, 162]. These methods use an *overlap-layout-consensus* approach, where reads are first aligned to each other (overlap), split into a number of paths representing the contigs (layout), and then the contigs are polished with a consensus of their reads. Some assemblers [47] skip the consensus step. Other graph-based methods which use concepts similar to overlap graphs or de Bruijn graphs but do not neatly fit into either category have also been published [163–167].

In our previous work [140], we used a sequence-to-graph alignment based hybrid method for diploid assembly. The method first used an external assembler to build a haploid assembly from the long reads. Then, a de Bruijn graph was built from short reads. The haploid assembly was aligned to the de Bruijn graph, and the reads were aligned to the graph as well. Then, bubbles in the graph were used to find arbitrary variation between the haplotypes. The bubbles were then phased based on the alignments of the reads, which split the haploid assembly into a diploid assembly. The main limitation of the previous project is that it requires an external assembler to first build the haploid reference.

In this section I describe a novel hybrid graph-based genome assembly pipeline. The pipeline is also based on sequence-to-graph alignment. The goal of the pipeline was to preserve genetic variation throughout the assembly process in the form of bubbles. Then, the bubbles could be used for phasing and for repeat resolution.

4.2.2 Methods

Figure 4.11 shows an overview of our genome assembly pipeline. The pipeline is essentially an overlap-layout-consensus approach without an explicit consensus step. However, the reads are implicitly error corrected during graph alignment and so the pipeline could be viewed as a "consensus-overlap-layout" approach.

First, a compacted de Bruijn graph is built from the short reads. The short reads are self-corrected using Lighter [121], and the graph is built using BCalm2 [67]. Next, long reads are aligned to the de Bruijn graph with GraphAligner [96]. The read sequence is replaced with the node IDs and orientations that the alignment passes through. The base pair sequences of the reads are at this point completely discarded. This essentially error corrects and compresses the reads. At this point the reads are represented by a list of node IDs (integers) with an orientation.

The reads in the node ID representation are then aligned to each other in an all-vs-all manner. This step requires a user given parameter for *minimum overlap match length*, which specifies the minimum number of matches in an alignment for it to be considered a valid overlap. All node IDs in the reads are weighted according to the node size in the compacted de Bruijn graph, and the weight of a match is equal to the node's length in the compacted de Bruijn graph. First, potential overlaps are filtered by checking for reads that share node IDs. A hash table is built with node IDs as keys, and reads as values. Then, pairs of reads that share node IDs are checked, and a potential overlap size between the two reads is calculated.

Given a list of number of occurrences of each node in the two reads occ^{read_1}, occ^{read_2} and a list of node lengths l, the potential overlap is calculated from $potential\_overlap = \Sigma_v \min(occ_v^{read_1}, occ_v^{read_2}) l_v$. The filtering does not check that the positions of the matches are consistent with an alignment, so the potential overlap is an upper bound on the number of matches between the two reads. Pairs whose potential overlap is less than the minimum overlap match length are then discarded, and those which might have an overlap longer than the minimum overlap match length are aligned.

The alignment is performed in a heuristic non-optimal manner. Given two reads, first the number of matches in each 150 bp diagonal band is counted. This estimation does not check that the matches form a valid alignment, and again is an upper bound on the number of matches. This approach has been used in sequence alignment to find potential match locations [168]. Then, the

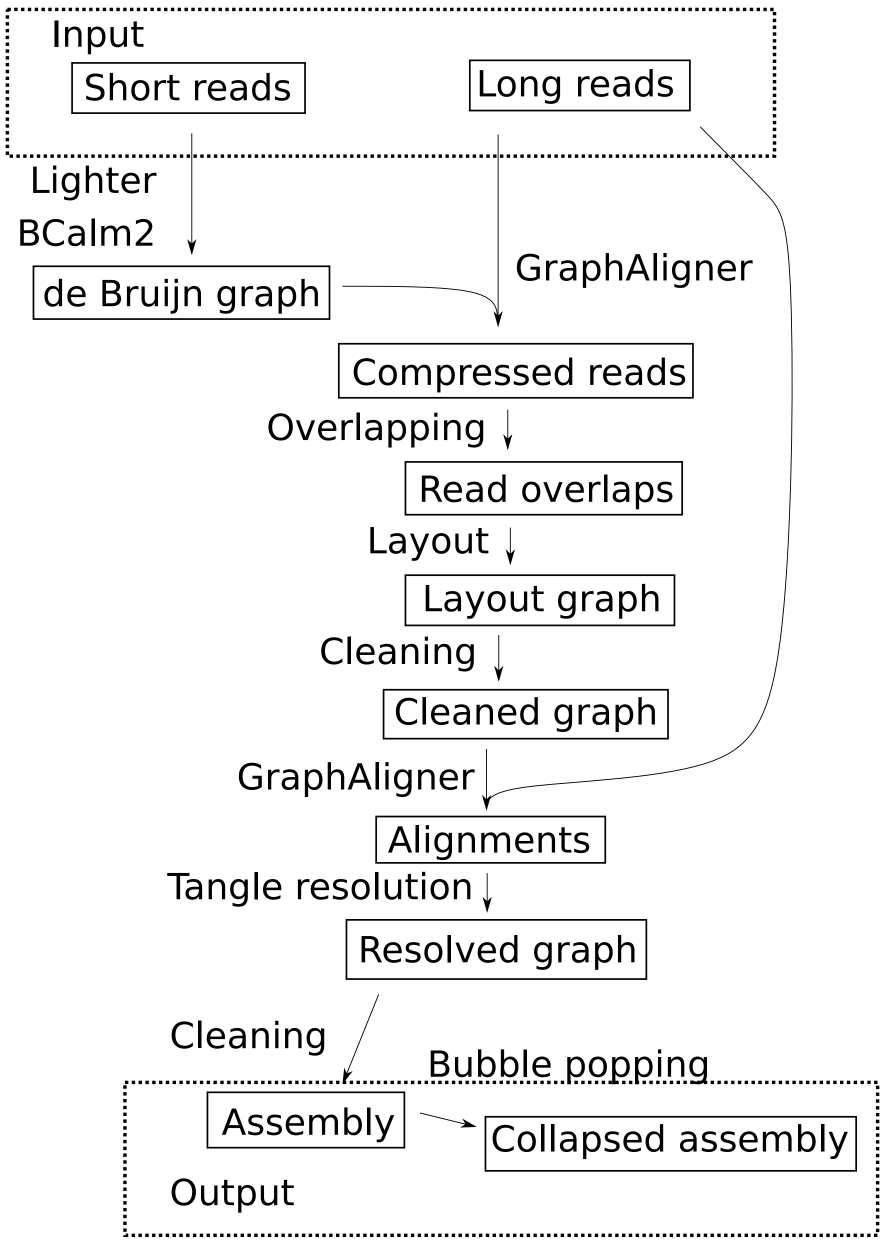


FIGURE 4.11 Overview of the genome assembly pipeline. The boxes with text represent data or files. The arrows represent the inputs of each step, and the labels next to the arrows describe which tool or process is used at that step. The input of the pipeline is a set of short reads and a set of long reads. First, the short reads are self-corrected with lighter [121] and assembled into a de Bruijn graph with bcalm2 [67]. Then, GraphAligner [96] is used to align the long reads into the de Bruijn graph. The long read sequences are then replaced with the sequence of node IDs of their alignment, essentially compressing and error correcting the reads. The alignments are used to induce overlaps between the long reads. The overlaps are then laid out into the layout graph. The layout graph is cleaned. Then, the long reads are re-aligned into the cleaned layout graph, and the alignments are used to resolve tangles. The tangle resolved graph is then cleaned to produce the assembly graph. Finally, a second *collapsed* assembly graph is created from the assembly graph by popping bubbles. The assembly graph contains bubbles which encode heterozygous positions, while the collapsed assembly is a haploid mosaic of the haplotypes.

150 bp band with the most potential matches is selected as the alignment band, and all matches which are in the band or in the neighbouring bands are eligible for alignment. The alignment is first initialized as an empty match. Then, instead of chaining the matches optimally, the matches are greedily added to the alignment. The matches are iterated in the order that they appear in the reads, and whenever a match can extend the current alignment, it is added to it. Algorithm 13 shows the pseudocode of the alignment. The runtime of this approach is $O(n)$ where n is the number of matches between the two reads. Since the reads are represented as a list of node IDs instead of base pairs, it is rare to have matches that are not part of the alignment, except in tandem repeats. At this point, alignments which contain less base pair matches than the minimum overlap match length are discarded.

The alignment is then extended with indels until it spans from the start of one of the reads to the end of one of the reads. This ensures that local alignments where only the middle parts of the reads match will contain a large amount of mismatches at the start and end.

Each alignment is finally scored according to the number of matches and mismatches in the alignment. Given a node v of length l_v, matches have a score of l_v, indels have a score of $-l_v$, and substitutions with a node u of length l_u have a score of $-\max(l_v, l_u)$. This corresponds to a score of $+1$ per matched base pair and -1 per mismatched base pair and indel.

Algorithm 13 Heuristic non-optimal read overlapping

```
1: Input: List of matches $S = (x_1, y_1, w_1), (x_2, y_2, w_2), ..., (x_n, y_n, w_n)$
2: where $x$ is the position on read 1, $y$ is the position of read 2 and $w$ is the length of the match
3: and read lengths $l_1$, $l_2$
4: Output: Alignment as a list of matches $L \subseteq S$.
5: $S$ is already sorted based on $x$, then based on $y$
6: $B \leftarrow$ an array with indices from $-\left\lceil \frac{l_2}{150} \right\rceil$ to $\left\lceil \frac{l_1}{150} \right\rceil$ initialized with 0
7: for $(x_i, y_i, w_i) \in S$ do
8:     $B[\left\lfloor \frac{x_i - y_i}{150} \right\rfloor] \leftarrow B[\left\lfloor \frac{x_i - y_i}{150} \right\rfloor] + w_i$
9: end for
10: $k \leftarrow i : B[i] = \max B$                ▷ Index with the highest value in B
11: $b_{min} \leftarrow k * 150 - 150$                ▷ Minimum diagonal of the band
12: $b_{max} \leftarrow k * 150 + 300$                ▷ Maximum diagonal of the band
13: for $(x_i, y_i, w_i) \in S$ do
14:     if $x_i - y_i > b_{min} \wedge x_i - y_i < b_{max}$ then
15:         if $L$ is empty $\vee (x_i > L.back().x \wedge y_i > L.back().y)$ then
16:             append($L, (x_i, y_i, w_i)$)
17:         end if
18:     end if
19: end for
20: return $L$
```

Given the list of all alignments, a subset of alignments are selected for the layout step. At each position in each read, the n highest scoring alignments are selected. Figure 4.12 shows the alignment selection with $n = 2$. Once the alignments have been selected for all reads, the nonselected alignments are discarded. Note that after the alignment selection step, the read can still be covered by more than n selected alignments at some points, as seen in Figure 4.12.

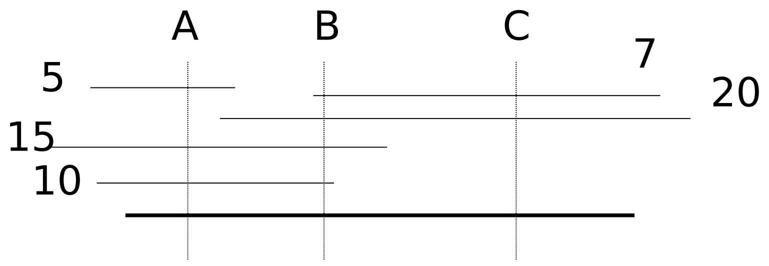


FIGURE 4.12 Alignment selection process. The thick line at the bottom represents a read. The thin lines above it are alignments with other reads. The number next to the alignment represents the alignment score. The dashed lines labeled A-C represent different positions along the read, which correspond to some base pair of the read. At each position in the read, the two highest scoring alignments which cover that position are selected. At position A, the alignments with scores 15 and 10 are selected, at B, the 20 and 15, and at C, the 20 and 7. Even though only the two highest scoring alignments are selected per position, some parts of the read can be covered by more than two selected alignments. For example, all four alignments that cover B are selected. The alignment with score 5 is not selected since each base pair covered by it is also covered by at least two other alignments with higher scores. All other alignments are selected.

Once the alignments have been selected, the reads are laid out into an assembly graph. Figure 4.13 shows the layout process. First, a graph is built where each read is represented by its node IDs, connected by edges. Then, the alignments are used to merge the nodes. Each alignment contains a list of pairs of matches. The matched pairs are merged into one node. Edges are also transferred to the new merged node. The final merged node will be the transitive closure over all nodes that were connected by an alignment match. A union-find data structure is used for merging the nodes.

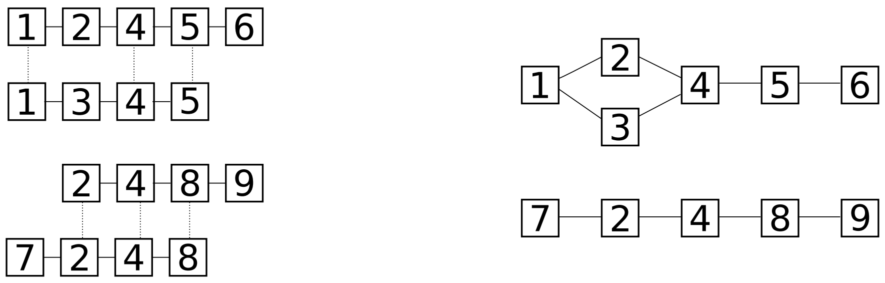


FIGURE 4.13 Graph layout. Left: four reads. Each row represents one read. The reads are represented as the node IDs of the de Bruijn graph. Each labeled square is one node within the read. The solid black lines are edges that connect the adjacent nodes within the read. The dashed lines are determined based on the selected alignments. Each match in an alignment connects the two matching nodes with a dashed line. Right: the resulting graph. The dashed lines are “squashed” such that the connected nodes are merged into one. There is an edge between two nodes if they were adjacent to each other in at least one read. The original de Bruijn graph node IDs can be duplicated if they were part of a repeat. For example, nodes 2 and 4, which represented a repeat in the de Bruijn graph, are now represented twice in the resulting graph, once per each repeat copy.

The assembly graph is then *cleaned*. Figure 4.14 shows the cleaning steps. First, nodes with low coverage are removed. Then, chimeric edges are removed. An edge is considered chimeric if its coverage is low, and one of the nodes connected by the edge has an another edge with significantly higher coverage. After this, tips are bridged. If a read passes through a tip and into an another tip, the read is used for connecting the two tips. This can happen if a linear area is cut due to having low coverage. Finally, short tips are removed from forks. When a fork leads into a short tip, but there is a much longer path starting at the fork, the short tip is removed. The cleaning step does not pop bubbles, unlike the cleaning steps in commonly used assemblers [47, 103, 162, 164]. The reason for this is to preserve heterozygosity in the assembly, which might help with phased assembly and might be more accurate than calling variants from the assembly afterwards.

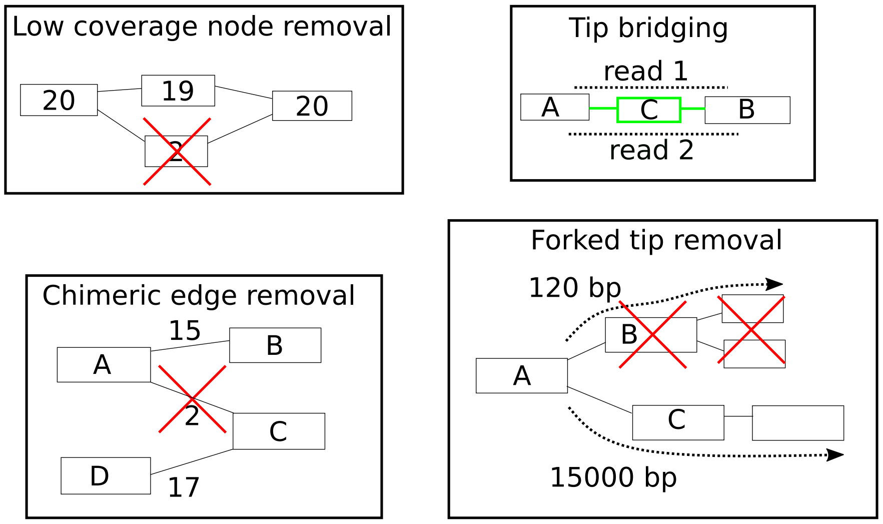


FIGURE 4.14 Graph cleaning. Low coverage node removal: nodes with a low coverage are removed. Chimeric edge removal: chimeric edges are removed. An edge is considered chimeric if it has low coverage and connects two nodes that have other, high coverage edges. The edge A-C is chimeric since it has a low coverage, and both A and C have other edges with high coverage, A-B and D-C. Tip bridging: tips are connected based on the reads. The nodes A and B are tips. However, reads 1 and 2 connect them. A new node C is added based on the sequences of the reads. Forked tip removal: short tips are removed at forks. At node A, there is a fork with two connecting nodes, node B and node C. The longest path starting at node B has a length of 120 bp, while the longest path starting at node C has a length of 15000 bp. Since the longest path starting at C is much longer, the entire branch starting at B is removed.

After the graph has been cleaned, small tangles are resolved. First, the raw long reads are all re-aligned to the assembly graph. Then, unique areas are called based on chains of superbubbles. A superbubble [73] is a structure in the graph consisting of a start node, an end node and a set of contained nodes. Assuming that all genomic sequences are represented in the graph, the number of genomic regions contained in the start and end nodes of the chain of superbubbles must then be equal throughout the entire chain. The copy count of the chain of superbubbles in the underlying genome can therefore be called.

The pipeline uses the coverage of the chain of superbubbles to define unique areas. The *cover-*

age of a chain of superbubbles is the average coverage along the start and end nodes in the chain. The *length* of a chain of superbubbles is the length of the shortest path from the start node to the end node. A chain of superbubbles is considered unique if its coverage is within 40% of the average long read coverage, and its length is at least 5kbp. Chains of superbubbles detected this way are virtually always unique, but some unique chains can be missed, especially chains which barely reach above the 5kbp length cutoff.

Given the unique areas of the graph and the alignments of the reads to the graph, the tangles are resolved. A *tangle* is a subgraph which contains only non-unique nodes, and which is connected to unique nodes only. A *bridging read* is a read whose alignment connects two unique areas through a tangle. Each unique area can have two sets of bridging reads, one for each end of the unique area. An end of a unique area is said to be *potentially resolvable* if at least 70% of its bridging reads connect to the same end of the same unique area, and unresolvable otherwise. If two ends of unique areas are potentially resolvable and 70% of reads on both ends connect to each other, then the two ends are *resolvable*. Given a pair of resolvable ends, the unique area is "cut out" of the tangle. Each edge connecting the resolvable ends to the tangle is removed. Then, the ends are connected based on the node IDs of the bridging reads. The bridging reads are aligned together using partial order alignment [80] and the resulting acyclic graph is inserted as the connection between the two ends. Note that since all reads used in the POA have the same start and end node ID, the resulting connection will be a chain of bubbles as well, so the two chains of bubbles end up being merged into one longer chain. If all of the ends connecting to a tangle are resolved, the tangle is also removed from the graph. Figure 4.15 shows the tangle resolution process.

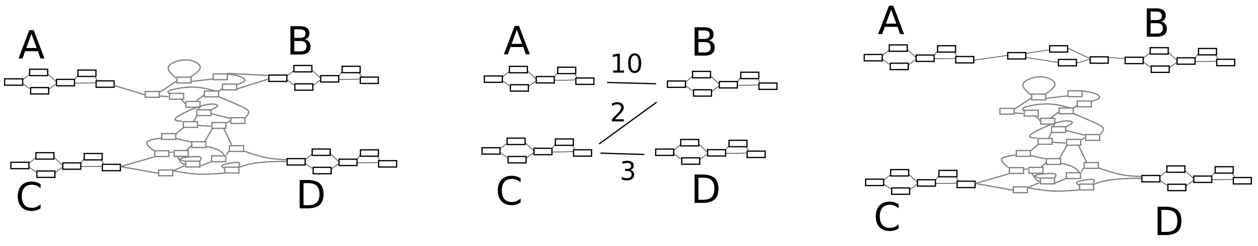


FIGURE 4.15 Tangle resolution. Left: Four unique areas (A, B, C, D) composed of a chain of bubbles each are shown in solid black. The gray area between the chains is a tangle, containing complicated structures and cycles. Middle: The tangle is ignored and the bridging reads are considered. 10 bridging reads connect A and B, 2 connect C and B and 3 connect C and D. A and B are resolvable with each other since 70% of bridging reads on both ends connect to each other. C is unresolvable since there is no connection with 70% of bridging reads. D is potentially resolvable with C, but since C is unresolvable, D stays unresolved. Right: A and B have been resolved. All edges connecting the tangle to A or B are removed, and the bridging reads are used to connect A and B, forming a longer chain of superbubbles. C and D remain connected to the tangle since they are unresolved. If C and D had been resolved, the tangle would have been removed from the graph as well.

The resolved assembly graph is then cleaned again using the same operations as before. The final output contains two graphs. One of the graphs is the assembly graph, preserving superbubbles

Genome	Long read coverage	NG50	NGA50	CPU-time
E. coli	30x	4.6 Mbp	4.6Mbp	20 min
S. cerevisiae	235x	810 kbp	740 kbp	5 h
D. melanogaster	32x	1.1 Mbp	-	17 h
H. sapiens HG00733	50x	58 kbp	-	350 h

TABLE 4.4 Results of the genome assembly pipeline

and thus variation. In the second graph, the superbubbles are popped and the sequence represents a mosaic of the haplotypes. The superbubbles are popped by arbitrarily picking one of the paths and removing all nodes and edges which are not covered by the path.

4.2.3 Results

Table 4.4 shows the results of applying the assembly pipeline to several datasets. Small genomes, such as the *E. coli* and *S. cerevisiae*, are assembled contiguously and the results are comparable with state-of-the art assemblers; the NGA50 is higher than the commonly used Canu [103] (version 1.8) for *S. cerevisiae*. However, longer genomes are fragmented. *D. melanogaster* has an NG50 of only 1.1 Mb. For the human HG00733, the assembly is very fragmented with an NG50 of only 58 kb. For comparison, other assemblers reach NGA50 values of multiple Gb for *D. melanogaster* and human with error-prone long reads [103, 162–164]. NGA50 was not evaluated for *D. melanogaster* and HG00733 since the NG50 was so low and NGA50 cannot be higher than NG50.

Table 4.5 shows the results for the graph-based assembly pipeline and Canu for *S. cerevisiae*. The graph-based pipeline achieved a higher NGA50 (740 kbp vs 552 kbp). Assembly sizes were similar to genome size for both assemblers. Despite including no consensus step, the base level error rate for the graph-based pipeline is comparable to Canu's. Canu had a slightly lower error rate for substitutions (10.75 vs 13.5), and about half of the error rate for indels (39.1 vs 82.4). The indel distribution is different between the two assemblers. Canu had a large amount of small indels (2485) while having few large indels (35). In contrast, the graph-based pipeline had fewer small indels (591) but more large indels (145). The higher indel error rate and the high number of large indels for the graph-based pipeline is likely due to gaps in short read coverage, which are currently not filled by using the long read sequences, and are instead left as deletions. The graph-based pipeline is over 5 times faster.

4.2.4 Discussion

The genome assembly pipeline uses a novel sequence-to-graph alignment based method. The pipeline assembles small genomes efficiently and accurately, even surpassing commonly used as-

	Graph-based	Canu
NGA50	740410	552014
Assembly size	12307952	12873840
Substitutions per 100kbp	13.5	10.75
Indels per 100kbp (bp)	82.4	39.1
N.o. indels $\leq$ 5bp	591	2485
N.o. indels > 5bp	145	35
CPU-time (h:mm:ss)	4:41:28	26:43:01
Peak memory (Gb)	6.5	6.9

TABLE 4.5 Results of the genome assembly pipeline for *S. cerevisiae* compared to Canu version 1.8

semblers for *S. cerevisiae*. However, the contiguity for the fruit fly genome is poor compared to commonly used assemblers, and the contiguity for the human genome assembly is orders of magnitude lower than expected. The reason for the low contiguity with more complex genomes is unknown.

The majority of the runtime is spent on aligning the long reads to the de Bruijn graph and overlapping them. In contrast, once the reads have been aligned and overlapped, the assembly itself takes only about 2,5% of the runtime for all of the genomes. Since the assembly takes a small fraction of the runtime for both the successful *S. cerevisiae* assembly as well as the failed *H. sapiens* assembly, it is unlikely that whatever is causing the poor contiguity in *H. sapiens* has a noticable effect on the runtime. It might therefore be possible to achieve accurate assemblies with a similarly quick runtime of about 350 cpu-hours for human assembly if the low contiguity is solved.

4.3 Conclusion

In this chapter I have presented two applications for GraphAligner. AERON quantifies RNA expression with long reads, achieving slightly better accuracy than linear alignment based quantification methods. AERON also detects fusion genes from long reads, a novel result that previous pipelines have not done.

The genome assembly pipeline uses a novel method for hybrid genome assembly. Small genomes are assembled accurately and efficiently. However, the genome assembly pipeline's results are not competitive with standard assemblers for complex genomes. The reason for the poor performance with human genomes is unknown.

The applications presented here show that GraphAligner can be used for various projects. As pangenomes become more common, the number of applications using sequence-to-graph alignment is likely to grow.

CHAPTER 5

Summary

In this work I have presented a theoretical basis for rapid sequence-to-graph alignment, the tool GraphAligner for aligning long reads to graphs in practice, and applications of GraphAligner. I have presented bit-parallel algorithms for aligning reads to genome graphs quickly, generalizing the Shift-And algorithm [20–23] and Myers' algorithm [9] for graphs. The bit-parallel sequence-to-graph alignment algorithm enables optimal alignment of long reads to bacterial sized genomes. This was not practical with previous algorithms.

I have also presented GraphAligner, a tool for aligning long reads to genome graphs rapidly. GraphAligner includes theoretical discoveries as well in the form of banded alignment to graphs. GraphAligner is focused on long reads due to the lack of tools for aligning long reads to graphs. Previous approaches have been unable to accurately and quickly align long reads to graphs. We are working on methods for extending it to short reads using the Pan-Genome Seeding Index [91]. GraphAligner can work on arbitrary graphs, including de Bruijn graphs with overlaps between nodes, and is faster than previous approaches by an order of magnitude in variation graphs as well as more accurate. GraphAligner enables fast and accurate error correction on human scale genomes and can be used along with vg [1] for genotyping variants.

I have also presented two applications of GraphAligner. AERON [97] is a pipeline for quantifying RNA expression and detecting fusion genes with long reads. AERON uses GraphAligner to align long reads to splicing graphs. The fusion gene detection uses a novel concept of fusion graphs, where aligning sequences to a graph is an essential part of the process. The fusion gene detection pipeline recovered known events in the K562 cancer cell line.

The genome assembly pipeline uses a novel hybrid graph-based assembly method. Short reads are used to build a de Bruijn graph, and long reads are aligned to the graph, error correcting and compressing the long reads. Runtime with small genomes is fast and the quality of the assembly is good. Unfortunately the assemblies do not exceed or match existing methods with complex genomes. If the pipeline could be made to work as well with complex genomes, it might deliver accurate assemblies quickly and cheaply.

GraphAligner has enabled sequence-to-graph alignment to scale to human sized genomes and outperforms existing tools. In this work I have presented some applications of GraphAligner, and as the field of pangenomics matures it is likely that GraphAligner will find wider use.

5.1 Future Work

Currently GraphAligner is suitable for aligning long reads. However, aligning short reads to genome graphs is also an important feature. The bit-parallel algorithm used by GraphAligner works just as well for short reads as long reads, however the implementation of the seed-and-extend method is currently not fit for short reads. The current method finds seed hits only within linear parts of the graph, not within areas containing large amounts of variation. Long reads can be aligned since they almost always cross simple linear parts. However, short reads might not be aligned into the complex areas. Since the entire point of using graphs is to include genetic variation, graph-based methods ought to be able to process areas with genetic variation, and being unable to align into complex areas defeats the point of using graphs in the first place. To resolve this, the Pan-Genome Seeding Index [91] can be used for seeding since it finds seeds everywhere in the graph.

The graph-based genome assembly pipeline currently has poor contiguity on complex genomes. The reason for the low contiguity is unknown. Fixing this issue might enable fast high-quality assembly of large genomes.

Combining polyploid phasing [169] with graph-based genome assembly is an interesting avenue for future development. A graph-based representation provides a simple and universal method for handling any kind of variation as bubbles in the graph. This would enable using complex structural variants for phasing as well instead of just SNPs.

Polyploid phasing on graphs is also related to repeat separation. Separating repeats is conceptually the same thing as polyploid phasing. The main difference is that in repeat separation the copy number might not be known exactly. Diploid phasing algorithms are typically used to "separate" haplotypes with more than 99.9% sequence identity. If polyploid phasing algorithms could be used for repeat separation, especially when combined with phasing of all SVs, not just SNPs, the contiguity of genome assemblies might be substantially increased.

The bit-parallel algorithm described in Chapter 2 can work on arbitrary graphs. Thus, it can also work with partial order alignment graphs [80] used for correcting reads and polishing assemblies. The bit-parallel method would likely lead to large speedups and enable rapid self-correction of long reads without requiring short reads.

The graph-based genotyping pipeline presented in Chapter 3 showed that GraphAligner and vg can be used for accurate genotyping with PacBio CCS reads. However, the pipeline uses the vg genotyping module, which is tuned for short reads. A module for graph-based genotyping tuned for long reads would likely improve the accuracy.

Recent papers [170, 171] have presented ways of speeding up optimal sequence-to-graph alignment. Combining these methods with GraphAligner would be an interesting avenue of development. The trie-based approach by Ivanov et al. [171] could be combined with GraphAligner. While optimal alignment is unlikely to reach the speed of heuristic seed-and-extend alignment, finding a guaranteed optimal alignment can still be useful in some applications. In particular, bacterial pangenomes are small enough that optimal alignment is practical.

Bibliography

[1] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, 2018.

[2] L. Salmela and E. Rivals, "Lordec: accurate and efficient long read error correction," *Bioinformatics*, vol. 30, no. 24, pp. 3506–3514, 2014.

[3] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10 915–10 919, 1992.

[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[5] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970.

[6] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[7] A. Backurs and P. Indyk, "Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)," in *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '15. New York, NY, USA: ACM, 2015, pp. 51–58.

[8] A. VL, E. DINITS, M. Kronrod, and F. IA, "On economical construction of transitive closure of an oriented graph," *Doklady Akademii Nauk SSSR*, vol. 194, no. 3, p. 487, 1970.

[9] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[10] E. Ukkonen, "Finding approximate patterns in strings," *Journal of Algorithms*, vol. 6, no. 1, pp. 132 – 137, 1985.

[11] A. Döring, D. Weese, T. Rausch, and K. Reinert, “Seqan an efficient, generic c++ library for sequence analysis,” *BMC bioinformatics*, vol. 9, no. 1, p. 11, 2008.

[12] J. Zhang, H. Lan, Y. Chan, Y. Shang, B. Schmidt, and W. Liu, “Bgsa: A bit-parallel global sequence alignment toolkit for multi-core and many-core architectures,” *Bioinformatics*, p. bty930, 2018. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bty930

[13] J. Loving, Y. Hernandez, and G. Benson, “Bitpal: a bit-parallel, general integer-scoring sequence alignment algorithm,” *Bioinformatics*, vol. 30, no. 22, pp. 3166–3173, 2014.

[14] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[15] E. W. Myers and W. Miller, “Optimal alignments in linear space,” *Bioinformatics*, vol. 4, no. 1, pp. 11–17, 1988.

[16] J. A. Grice, R. Hughey, and D. Speck, “Reduced space sequence alignment,” *Bioinformatics*, vol. 13, no. 1, pp. 45–53, 1997.

[17] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[18] P. H. Sellers, “The theory and computation of evolutionary distances: Pattern recognition,” *J. Algorithm. Comput. Technol.*, vol. 1, no. 4, pp. 359–373, Dec. 1980.

[19] E. Ukkonen, “Algorithms for approximate string matching,” *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.

[20] B. Dömölki, “An algorithm for syntactical analysis,” *Computational Linguistics*, vol. 3, no. 29-46, p. 151, 1964.

[21] ——, “A universal compiler system based on production rules,” *BIT Numerical Mathematics*, vol. 8, no. 4, pp. 262–275, 1968.

[22] R. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *Commun. ACM*, vol. 35, no. 10, pp. 74–82, Oct. 1992.

[23] R. Baeza-Yates and G. Navarro, “A faster algorithm for approximate string matching,” in *Combinatorial Pattern Matching*, D. Hirschberg and G. Myers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–23.

[24] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 390–398.

[25] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Bioinformatics*, vol. 8, no. 5, pp. 481–487, 1992.

[26] H. Suzuki and M. Kasahara, "Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming," *bioRxiv*, 2017.

[27] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, "Mummer4: A fast and versatile genome alignment system," *PLoS computational biology*, vol. 14, no. 1, p. e1005944, 2018.

[28] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[29] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[30] I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, "Fast and sensitive mapping of nanopore sequencing reads with graphmap," *Nature communications*, vol. 7, p. 11307, 2016.

[31] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, 1973, pp. 1–11.

[32] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[33] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[34] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *International colloquium on automata, languages, and programming*. Springer, 2003, pp. 943–955.

[35] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of discrete algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[36] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[37] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[38] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[39] C. Jain, A. Dilthey, S. Koren, S. Aluru, and A. M. Phillippy, “A fast approximate algorithm for mapping long reads to large reference databases,” in *International Conference on Research in Computational Molecular Biology*. Springer, 2017, pp. 66–81.

[40] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[41] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, “Reducing storage requirements for biological sequence comparison,” *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[42] G. Marçais, D. Pellow, D. Bork, Y. Orenstein, R. Shamir, and C. Kingsford, “Improving the performance of minimizers and winnowing schemes,” *Bioinformatics*, vol. 33, no. 14, pp. i110–i117, 2017.

[43] A. L. Delcher, S. L. Salzberg, and A. M. Phillippy, “Using mummer to identify similar regions in large sequence sets,” *Current protocols in bioinformatics*, no. 1, pp. 10–3, 2003.

[44] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, “Ultrafast and memory-efficient alignment of short dna sequences to the human genome,” *Genome biology*, vol. 10, no. 3, p. R25, 2009.

[45] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with bowtie 2,” *Nature methods*, vol. 9, no. 4, p. 357, 2012.

[46] M. J. Chaisson and G. Tesler, “Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory,” *BMC bioinformatics*, vol. 13, no. 1, p. 238, 2012.

[47] H. Li, “Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences,” *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[48] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru, “A fast adaptive algorithm for computing whole-genome homology maps,” *Bioinformatics*, vol. 34, no. 17, pp. i748–i756, 2018.

[49] G. Myers and W. Miller, “Chaining multiple-alignment fragments in sub-quadratic time,” in *SODA*, vol. 95, 1995, pp. 38–47.

[50] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu, *Genome-scale algorithm design*. Cambridge University Press, 2015.

[51] F. J. Sedlazeck, B. Yu, A. J. Mansfield, H. Chen, O. Krasheninina, A. Tin, Q. Qi, S. Zarate, J. Traynelis, V. Menon, , J. Hu, h. v. doddapaneni, G. Metcalf, J. Coresh, R. Kaplan, d. m. muzny, G. Jun, R. A. Gibbs, W. Salerno, and E. Boerwinkle, “Multiethnic catalog of structural variants and their translational impact for disease phenotypes across 19,652 genomes,” *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/05/03/2020.05.02.074096

[52] J. O. Korbel, A. E. Urban, J. P. Affourtit, B. Godwin, F. Grubert, J. F. Simons, P. M. Kim, D. Palejev, N. J. Carriero, L. Du *et al.*, “Paired-end mapping reveals extensive structural variation in the human genome,” *Science*, vol. 318, no. 5849, pp. 420–426, 2007.

[53] R. Redon, S. Ishikawa, K. R. Fitch, L. Feuk, G. H. Perry, T. D. Andrews, H. Fiegler, M. H. Shapero, A. R. Carson, W. Chen *et al.*, “Global variation in copy number in the human genome,” *nature*, vol. 444, no. 7118, pp. 444–454, 2006.

[54] J. Sebat, B. Lakshmi, J. Troge, J. Alexander, J. Young, P. Lundin, S. Månér, H. Massa, M. Walker, M. Chi *et al.*, “Large-scale copy number polymorphism in the human genome,” *Science*, vol. 305, no. 5683, pp. 525–528, 2004.

[55] I. H. Consortium *et al.*, “A haplotype map of the human genome,” *Nature*, vol. 437, no. 7063, p. 1299, 2005.

[56] M. J. Chaisson, A. D. Sanders, X. Zhao, A. Malhotra, D. Porubsky, T. Rausch, E. J. Gardner, O. L. Rodriguez, L. Guo, R. L. Collins *et al.*, “Multi-platform discovery of haplotype-resolved structural variation in human genomes,” *Nature communications*, vol. 10, 2019.

[57] R. M. Sherman, J. Forman, V. Antonescu, D. Puiu, M. Daya, N. Rafaels, M. P. Boorgula, S. Chavan, C. Vergara, V. E. Ortega *et al.*, “Assembly of a pan-genome from deep sequencing of 910 humans of african descent,” *Nature genetics*, vol. 51, no. 1, pp. 30–35, 2019.

[58] B. E. Stranger, M. S. Forrest, M. Dunning, C. E. Ingle, C. Beazley, N. Thorne, R. Redon, C. P. Bird, A. De Grassi, C. Lee *et al.*, “Relative impact of nucleotide and copy number variation on gene expression phenotypes,” *Science*, vol. 315, no. 5813, pp. 848–853, 2007.

[59] J. L. Freeman, G. H. Perry, L. Feuk, R. Redon, S. A. McCarroll, D. M. Altshuler, H. Aburatani, K. W. Jones, C. Tyler-Smith, M. E. Hurles *et al.*, “Copy number variation: new insights in genome diversity,” *Genome research*, vol. 16, no. 8, pp. 949–961, 2006.

[60] “Computational pan-genomics: status, promises and challenges,” *Briefings in bioinformatics*, vol. 19, no. 1, pp. 118–135, 2016.

[61] D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, N. Bouk, H.-C. Chen, R. Agarwala, W. M. McLaren, G. R. Ritchie *et al.*, “Modernizing reference genome assemblies,” *PLoS biology*, vol. 9, no. 7, 2011.

[62] A. Danek, S. Deorowicz, and S. Grabowski, “Indexes of large genome collections on a pc,” *PloS one*, vol. 9, no. 10, p. e109384, 2014.

[63] R. Rahn, D. Weese, and K. Reinert, “Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop,” *Bioinformatics*, vol. 30, no. 24, pp. 3499–3505, 2014.

[64] R. Durbin, “Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt),” *Bioinformatics*, vol. 30, no. 9, pp. 1266–1272, 2014.

[65] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison, “Genome graphs and the evolution of genome inference,” *Genome research*, vol. 27, no. 5, pp. 665–676, 2017.

[66] P. A. Pevzner, H. Tang, and M. S. Waterman, “An eulerian path approach to dna fragment assembly,” *Proceedings of the national academy of sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[67] R. Chikhi, A. Limasset, and P. Medvedev, “Compacting de bruijn graphs from sequencing data quickly and in low memory,” *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[68] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski *et al.*, “Spades: a new genome assembly algorithm and its applications to single-cell sequencing,” *Journal of computational biology*, vol. 19, no. 5, pp. 455–477, 2012.

[69] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, “De novo assembly and genotyping of variants using colored de bruijn graphs,” *Nature genetics*, vol. 44, no. 2, p. 226, 2012.

[70] F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro, “A space and time-efficient index for the compacted colored de bruijn graph,” *Bioinformatics*, vol. 34, no. 13, pp. i169–i177, 2018.

[71] G. Holley and P. Melsted, “Bifrost–highly parallel construction and indexing of colored and compacted de bruijn graphs,” *BioRxiv*, p. 695338, 2019.

[72] C. Marchet, C. Boucher, S. J. Puglisi, P. Medvedev, M. Salson, and R. Chikhi, “Data structures based on k-mers for querying large collections of sequencing datasets,” *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/12/06/866756

[73] T. Onodera, K. Sadakane, and T. Shibuya, “Detecting superbubbles in assembly graphs,” in *International Workshop on Algorithms in Bioinformatics*. Springer, 2013, pp. 338–348.

[74] B. Paten, J. M. Eizenga, Y. M. Rosen, A. M. Novak, E. Garrison, and G. Hickey, “Superbubbles, ultrabubbles, and cacti,” *Journal of Computational Biology*, vol. 25, no. 7, pp. 649–663, 2018.

[75] G. Hickey, D. Heller, J. Monlong, J. A. Sibbesen, J. Siren, J. Eizenga, E. Dawson, E. Garrison, A. Novak, and B. Paten, “Genotyping structural variants in pangenome graphs using the vg toolkit,” *BioRxiv*, p. 654566, 2019.

[76] J. R. Wang, J. Holt, L. McMillan, and C. D. Jones, “Fmlrc: Hybrid long read error correction using an fm-index,” *BMC bioinformatics*, vol. 19, no. 1, p. 50, 2018.

[77] E. W. Myers and W. Miller, “Approximate matching of regular expressions,” *Bulletin of Mathematical Biology*, vol. 51, no. 1, pp. 5 – 37, 1989.

[78] G. Navarro, “Improved approximate pattern matching on hypertext,” *Theoretical Computer Science*, vol. 237, no. 1, pp. 455 – 463, 2000.

[79] M. Equi, R. Grossi, A. I. Tomescu, and V. Mäkinen, “On the complexity of exact pattern matching in graphs: Determinism and zig-zag matching,” *arXiv preprint arXiv:1902.03560*, 2019.

[80] C. Lee, C. Grasso, and M. F. Sharlow, “Multiple sequence alignment using partial order graphs,” *Bioinformatics*, vol. 18, no. 3, pp. 452–464, 2002.

[81] V. N. S. Kavya, K. Tayal, R. Srinivasan, and N. Sivadasan, “Sequence alignment on directed graphs,” *Journal of Computational Biology*, vol. 26, no. 1, pp. 53–67, 2019.

[82] A. Limasset, B. Cazaux, E. Rivals, and P. Peterlongo, “Read mapping on de bruijn graphs,” *BMC Bioinformatics*, vol. 17, no. 1, p. 237, 16 Jun. 2016.

[83] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, “hybridspades: an algorithm for hybrid assembly of short and long reads,” *Bioinformatics*, vol. 32, no. 7, pp. 1009–1015, 2015.

[84] H. Li, “minigraph,” https://github.com/lh3/minigraph, 2019.

[85] M. Rautiainen and T. Marschall, “Aligning sequences to general graphs in o(v + me) time,” *bioRxiv*, 2017.

[86] C. Jain, H. Zhang, Y. Gao, and S. Aluru, “On the complexity of sequence to graph alignment,” *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/01/17/522912

[87] T. Dvorkina, D. Antipov, A. Korobeynikov, and S. Nurk, “Spaligner: alignment of long diverged molecular sequences to assembly graphs,” *BioRxiv*, p. 744755, 2019.

[88] M. Equi, V. Mäkinen, and A. I. Tomescu, “Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless seth fails,” 2020.

[89] J. Sirén, “Indexing variation graphs,” in *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*. SIAM, 2017, pp. 13–27.

[90] J. Sirén, E. Garrison, A. M. Novak, B. Paten, and R. Durbin, “Haplotype-aware graph indexes,” *arXiv preprint arXiv:1805.03834*, 2018.

[91] A. Ghaffaari and T. Marschall, “Fully-sensitive seed finding in sequence graphs using a hybrid index,” in *International Conference on Research in Computational Molecular Biology*, 2019.

[92] T. Mokveld, J. Linthorst, Z. Al-Ars, H. Holstege, and M. Reinders, “Chop: Haplotype-aware path indexing in population graphs,” *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/03/14/305268

[93] T. Gagie, G. Manzini, and J. Sirén, “Wheeler graphs: A framework for bwt-based data structures,” *Theoretical computer science*, vol. 698, pp. 67–78, 2017.

[94] J. Alanko, A. Policriti, and N. Prezza, “On prefix-sorting finite automata,” *arXiv preprint arXiv:1902.01088*, 2019.

[95] M. Rautiainen, V. Mäkinen, and T. Marschall, “Bit-parallel sequence-to-graph alignment,” *Bioinformatics*, 03 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz162

[96] M. Rautiainen and T. Marschall, “Graphaligner: Rapid and versatile sequence-to-graph alignment,” *BioRxiv*, p. 810812, 2019.

[97] M. Rautiainen, D. A. Durai, Y. Chen, L. Xin, H. M. Low, J. Göke, T. Marschall, and M. H. Schulz, “Aeron: Transcript quantification and gene-fusion detection using long reads,” *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/01/27/2020.01.27.921338

[98] R. R. Wick, M. B. Schultz, J. Zobel, and K. E. Holt, “Bandage: interactive visualization of de novo genome assemblies,” *Bioinformatics*, vol. 31, no. 20, pp. 3350–3352, 2015.

[99] Y. Ono, K. Asai, and M. Hamada, “Pbsim: Pacbio reads simulator-toward accurate genome assembly,” *Bioinformatics*, vol. 29, no. 1, pp. 119–121, 2013.

[100] J. Robinson, J. A. Halliwell, J. D. Hayhurst, P. Flicek, P. Parham, and S. G. E. Marsh, “The IPD and IMGT/HLA database: allele variant databases,” *Nucleic Acids Res.*, vol. 43, no. Database issue, pp. D423–31, Jan. 2015.

[101] F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding *et al.*, “Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega,” *Molecular systems biology*, vol. 7, no. 1, p. 539, 2011.

[102] D. J. Lipman and W. R. Pearson, “Rapid and sensitive protein similarity searches,” *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985.

[103] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, “Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation,” *Genome research*, pp. gr–215 087, 2017.

[104] S. D. Jackman, B. P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S. A. Hammond, G. Jahesh, H. Khan, L. Coombe, R. L. Warren *et al.*, “Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter,” *Genome research*, vol. 27, no. 5, pp. 768–777, 2017.

[105] J. Edmonds and E. L. Johnson, “Matching: A well-solved class of integer linear programs,” in *Combinatorial Optimization—Eureka, You Shrink!* Springer, 2003, pp. 27–30.

[106] P. Medvedev and M. Brudno, “Maximum likelihood genome assembly,” *Journal of computational Biology*, vol. 16, no. 8, pp. 1101–1116, 2009.

[107] E. W. Myers, “The fragment assembly string graph,” *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[108] J. Sirén, N. Välimäki, and V. Mäkinen, “Indexing graphs for path queries with applications in genome research,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 11, no. 2, pp. 375–388, 2014.

[109] S. Gog, T. Beller, A. Moffat, and M. Petri, “From theory to practice: Plug and play with succinct data structures,” in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337.

[110] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo, “Fast and scalable minimal perfect hashing for massive key sets,” *arXiv preprint arXiv:1702.03154*, 2017.

[111] A. Kuosmanen, T. Paavilainen, T. Gagie, R. Chikhi, A. Tomescu, and V. Mäkinen, “Using minimum path cover to boost dynamic programming on dags: co-linear chaining extended,” in *International Conference on Research in Computational Molecular Biology*. Springer, 2018, pp. 105–121.

[112] Z. Zhang, P. Berman, T. Wiehe, and W. Miller, “Post-processing long pairwise alignments,” *Bioinformatics*, vol. 15, no. 12, pp. 1012–1019, 1999.

[113] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[114] L. Clarke, S. Fairley, X. Zheng-Bradley, I. Streeter, E. Perry, E. Lowy, A.-M. Tassé, and P. Flicek, “The international genome sample resource (igsr): A worldwide collection of genome variation incorporating the 1000 genomes project data,” *Nucleic acids research*, vol. 45, no. D1, pp. D854–D859, 2017.

[115] A. M. Wenger, P. Peluso, W. J. Rowell, P.-C. Chang, R. J. Hall, G. T. Concepcion, J. Ebler, A. Fungtammasan, A. Kolesnikov, N. D. Olson *et al.*, “Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome,” *Nature biotechnology*, vol. 37, no. 10, pp. 1155–1162, 2019.

[116] P. Krusche, L. Trigg, P. C. Boutros, C. E. Mason, M. Francisco, B. L. Moore, M. Gonzalez-Porta, M. A. Eberle, Z. Tezak, S. Lababidi *et al.*, “Best practices for benchmarking germline small-variant calls in human genomes,” *Nature biotechnology*, vol. 37, no. 5, pp. 555–560, 2019.

[117] E. Lowy-Gallego, S. Fairley, X. Zheng-Bradley, M. Ruffier, L. Clarke, P. Flicek, . G. P. Consortium *et al.*, “Variant calling on the grch38 assembly with the data from phase three of the 1000 genomes project,” *Wellcome Open Research*, vol. 4, 2019.

[118] J. G. Cleary, R. Braithwaite, K. Gaastra, B. S. Hilbush, S. Inglis, S. A. Irvine, A. Jackson, R. Littin, M. Rathod, D. Ware *et al.*, “Comparing variant call files for performance benchmarking of next-generation sequencing variant calling pipelines,” *BioRxiv*, p. 023754, 2015.

[119] G. Miclotte, M. Heydari, P. Demeester, S. Rombauts, Y. Van de Peer, P. Audenaert, and J. Fostier, “Jabba: hybrid error correction for long sequencing reads,” *Algorithms for Molecular Biology*, vol. 11, no. 1, p. 10, 2016.

[120] H. Zhang, C. Jain, and S. Aluru, “A comprehensive evaluation of long read error correction methods,” *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/01/13/519330

[121] L. Song, L. Florea, and B. Langmead, “Lighter: fast and memory-efficient sequencing error correction without counting,” *Genome biology*, vol. 15, no. 11, p. 509, 2014.

[122] J. Holt and L. McMillan, “Merging of multi-string bwts with applications,” *Bioinformatics*, vol. 30, no. 24, pp. 3524–3531, 2014.

[123] H. Li, “Fast construction of fm-index for long sequence reads,” *Bioinformatics*, vol. 30, no. 22, pp. 3274–3275, 2014.

[124] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, “The sequence alignment/map format and samtools,” *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[125] J. L. Weirather, M. de Cesare, Y. Wang, P. Piazza, V. Sebastiano, X.-J. Wang, D. Buck, and K. F. Au, “Comprehensive comparison of pacific biosciences and oxford nanopore technologies and their applications to transcriptome analysis,” *F1000Research*, vol. 6, 2017.

[126] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, “The Cancer Genome Atlas Pan-Cancer Analysis Project,” *Nature Genetics*, October 2013.

[127] N. S. Latysheva and M. M. Babu, “Discovering and understanding oncogenic gene fusions through data intensive computational approaches,” *Nucleic Acids Research*, vol. 44, no. 10, p. 4487–4503, 2016.

[128] E. T. Wang, R. Sandberg, S. Luo, I. Khrebtukova, L. Zhang, C. Mayr, S. F. Kingsmore, G. P. Schroth, and C. B. Burge, “Alternative isoform regulation in human tissue transcriptomes,” *Nature*, vol. 456, no. 7221, p. 470–476, 2008.

[129] Y. Huang, Y. Hu, D. J. Corbin, N. J. MacLeod, D. Y. Chiang, Y. Liu, J. F. Prins, and J. Li, “A Robust Method for Transcript Quantification with RNA-Seq Data,” *Journal of Computational Biology*, 2013.

[130] S. Karlin and S. F. Altschul, “Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes,” *Proceedings of the National Academy of Sciences*, vol. 87, no. 6, pp. 2264–2268, 1990.

[131] R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford, “Salmon: fast and bias-aware quantification of transcript expression using dual-phase inference,” *Nature Methods*, 2017.

[132] B. Zhou, S. S. Ho, S. U. Greer, X. Zhu, J. M. Bell, J. G. Arthur, N. Spies, X. Zhang, S. Byeon, R. Pattni, N. Ben-Efraim, M. S. Haney, R. R. Haraksingh, G. Song, H. P. Ji, D. Perrin, W. H. Wong, A. Abyzov, and A. E. Urban, “Comprehensive, integrated, and phased whole-genome analysis of the primary ENCODE cell line K562,” *Genome Res.*, Feb. 2019.

[133] H. Thorvaldsdóttir, J. T. Robinson, and J. P. Mesirov, “Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration,” *Briefings in bioinformatics*, vol. 14, no. 2, pp. 178–192, 2013.

[134] S.-T. Lee, E.-H. Yoo, J.-Y. Kim, J.-W. Kim, and C.-S. Ki, “Multiplex ligation-dependent probe amplification screening of isolated increased HbF levels revealed three cases of novel

rearrangements/deletions in the beta-globin gene cluster," *British Journal of Haematology*, vol. 148, no. 1, pp. 154–160, Jan. 2010.

[135] R. Kurzrock, H. M. Kantarjian, B. J. Druker, and M. Talpaz, "Philadelphia chromosome-positive leukemias: from basic mechanisms to molecular therapeutics." *Annals of Internal Medicine*, pp. 819–830, 2003.

[136] T. Prakash, V. Sharma, N. Adati, R. Ozawa, N. Kumar, Y. Nishida, T. Fujikake, T. Takeda, and T. Taylor, "Expression of conjoined genes: Another mechanism for gene regulation in eukaryotes," *PLoS One*, vol. 5, 2010.

[137] M. G. C. M. P. Team *et al.*, "Generation and initial analysis of more than 15,000 full-length human and mouse cDNA sequences," *Proceedings of the National Academy of Sciences*, vol. 99, no. 26, pp. 16 899–16 903, 2002.

[138] G. K. C. of Scientists, "Genome 10k: a proposal to obtain whole-genome sequence for 10 000 vertebrate species," *Journal of Heredity*, vol. 100, no. 6, pp. 659–674, 2009.

[139] "Vertebrate genomes project," https://vertebrategenomesproject.org/, accessed: 2020-04-29.

[140] S. Garg, M. Rautiainen, A. M. Novak, E. Garrison, R. Durbin, and T. Marschall, "A graph-based approach to diploid genome assembly," *Bioinformatics*, vol. 34, no. 13, pp. i105–i114, 2018.

[141] S. Koren, A. Rhie, B. P. Walenz, A. T. Dilthey, D. M. Bickhart, S. B. Kingan, S. Hiendleder, J. L. Williams, T. P. Smith, and A. M. Phillippy, "De novo assembly of haplotype-resolved genomes with trio binning," *Nature biotechnology*, vol. 36, no. 12, pp. 1174–1182, 2018.

[142] Z. N. Kronenberg, R. J. Hall, S. Hiendleder, T. P. Smith, S. T. Sullivan, J. L. Williams, and S. B. Kingan, "Falcon-phase: integrating pacbio and hi-c data for phased diploid genomes," *Biorxiv*, p. 327064, 2018.

[143] D. Porubsky, P. Ebert, P. A. Audano, M. R. Vollger, W. T. Harvey, K. M. Munson, M. Sorensen, A. Sulovari, M. Haukness, M. Ghareghani *et al.*, "A fully phased accurate assembly of an individual human genome," *bioRxiv*, p. 855049, 2019.

[144] C.-S. Chin and A. Khalak, "Human genome assembly in 100 minutes," *bioRxiv*, p. 705616, 2019.

[145] S. Nurk, B. P. Walenz, A. Rhie, M. R. Vollger, G. A. Logsdon, R. Grothe, K. H. Miga, E. E. Eichler, A. M. Phillippy, and S. Koren, "Hicanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads," *bioRxiv*, 2020.

[146] "hifiasm," https://github.com/chhylp123/hifiasm, accessed: 2020-04-29.

[147] K. H. Miga, S. Koren, A. Rhie, M. R. Vollger, A. Gershman, A. Bzikadze, S. Brooks, E. Howe, D. Porubsky, G. A. Logsdon *et al.*, “Telomere-to-telomere assembly of a complete human x chromosome,” *BioRxiv*, p. 735928, 2019.

[148] S. Zhou, J. Herschleb, and D. C. Schwartz, “A single molecule system for whole genome analysis,” *Perspectives in Bioanalysis*, vol. 2, pp. 265–300, 2007.

[149] E. K. Chan, D. L. Cameron, D. C. Petersen, R. J. Lyons, B. F. Baldi, A. T. Papenfuss, D. M. Thomas, and V. M. Hayes, “Optical mapping reveals a higher level of genomic architecture of chained fusions in cancer,” *Genome research*, vol. 28, no. 5, pp. 726–738, 2018.

[150] G. X. Zheng, B. T. Lau, M. Schnall-Levin, M. Jarosz, J. M. Bell, C. M. Hindson, S. Kyriazopoulou-Panagiotopoulou, D. A. Masquelier, L. Merrill, J. M. Terry *et al.*, “Haplotyping germline and cancer genomes with high-throughput linked-read sequencing,” *Nature biotechnology*, vol. 34, no. 3, p. 303, 2016.

[151] E. Lieberman-Aiden, N. L. Van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner *et al.*, “Comprehensive mapping of long-range interactions reveals folding principles of the human genome,” *science*, vol. 326, no. 5950, pp. 289–293, 2009.

[152] J. Ghurye, A. Rhie, B. P. Walenz, A. Schmitt, S. Selvaraj, M. Pop, A. M. Phillippy, and S. Koren, “Integrating hi-c links with assembly graphs for chromosome-scale assembly,” *PLoS computational biology*, vol. 15, no. 8, p. e1007273, 2019.

[153] E. Falconer, M. Hills, U. Naumann, S. S. Poon, E. A. Chavez, A. D. Sanders, Y. Zhao, M. Hirst, and P. M. Lansdorp, “Dna template strand sequencing of single-cells maps genomic rearrangements at high resolution,” *Nature methods*, vol. 9, no. 11, pp. 1107–1112, 2012.

[154] E. Falconer and P. M. Lansdorp, “Strand-seq: a unifying tool for studies of chromosome segregation,” in *Seminars in cell & developmental biology*, vol. 24, no. 8-9. Elsevier, 2013, pp. 643–652.

[155] D. Porubskỳ, A. D. Sanders, N. Van Wietmarschen, E. Falconer, M. Hills, D. C. Spierings, M. R. Bevova, V. Guryev, and P. M. Lansdorp, “Direct chromosome-length haplotyping by single-cell sequencing,” *Genome research*, vol. 26, no. 11, pp. 1565–1574, 2016.

[156] D. Porubsky, S. Garg, A. D. Sanders, J. O. Korbel, V. Guryev, P. M. Lansdorp, and T. Marschall, “Dense and accurate whole-chromosome haplotyping of individual genomes,” *Nature communications*, vol. 8, no. 1, pp. 1–10, 2017.

[157] M. Ghareghani, D. Porubskỳ, A. D. Sanders, S. Meiers, E. E. Eichler, J. O. Korbel, and T. Marschall, “Strand-seq enables reliable separation of long reads by chromosome via expectation maximization,” *Bioinformatics*, vol. 34, no. 13, pp. i115–i123, 2018.

[158] B. Ewing, L. Hillier, M. C. Wendl, and P. Green, “Base-calling of automated sequencer traces usingphred. i. accuracy assessment,” *Genome research*, vol. 8, no. 3, pp. 175–185, 1998.

[159] F. A. Simão, R. M. Waterhouse, P. Ioannidis, E. V. Kriventseva, and E. M. Zdobnov, “Busco: assessing genome assembly and annotation completeness with single-copy orthologs,” *Bioinformatics*, vol. 31, no. 19, pp. 3210–3212, 2015.

[160] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, “Quast: quality assessment tool for genome assemblies,” *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, 2013.

[161] R. R. Wick, L. M. Judd, C. L. Gorrie, and K. E. Holt, “Unicycler: resolving bacterial genome assemblies from short and long sequencing reads,” *PLoS computational biology*, vol. 13, no. 6, p. e1005595, 2017.

[162] C.-S. Chin, P. Peluso, F. J. Sedlazeck, M. Nattestad, G. T. Concepcion, A. Clum, C. Dunn, R. O’Malley, R. Figueroa-Balderas, A. Morales-Cruz *et al.*, “Phased diploid genome assembly with single-molecule real-time sequencing,” *Nature methods*, vol. 13, no. 12, p. 1050, 2016.

[163] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, “Assembly of long, error-prone reads using repeat graphs,” *Nature biotechnology*, vol. 37, no. 5, pp. 540–546, 2019.

[164] J. Ruan and H. Li, “Fast and accurate long-read assembly with wtdbg2,” *Nature Methods*, pp. 1–4, 2019.

[165] R. Kajitani, D. Yoshimura, M. Okuno, Y. Minakuchi, H. Kagoshima, A. Fujiyama, K. Kubokawa, Y. Kohara, A. Toyoda, and T. Itoh, “Platanus-allee is a de novo haplotype assembler enabling a comprehensive access to divergent heterozygous regions,” *Nature communications*, vol. 10, no. 1, pp. 1–15, 2019.

[166] G. M. Kamath, I. Shomorony, F. Xia, T. A. Courtade, and N. T. David, “Hinge: long-read assembly achieves optimal repeat resolution,” *Genome research*, vol. 27, no. 5, pp. 747–756, 2017.

[167] K. Shafin, T. Pesout, R. Lorig-Roach, M. Haukness, H. E. Olsen, C. Bosworth, J. Armstrong, K. Tigyi, N. Maurer, S. Koren *et al.*, “Efficient de novo assembly of eleven human genomes using promethion sequencing and a novel nanopore toolkit,” *BioRxiv*, p. 715722, 2019.

[168] K. R. Rasmussen, J. Stoye, and E. W. Myers, “Efficient q-gram filters for finding all ε-matches over a given length,” *Journal of Computational Biology*, vol. 13, no. 2, pp. 296–308, 2006.

[169] S. Schrinner, R. S. Mari, J. W. Ebler, M. Rautiainen, L. Seillier, J. Reimer, B. Usadel, T. Marschall, and G. W. Klau, “Haplotype threading: Accurate polyploid phasing from long reads,” *BioRxiv*, 2020.

[170] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru, “Accelerating sequence alignment to graphs,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 451–461.

[171] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, and M. Vechev, “Astarix: Fast and optimal sequence-to-graph alignment,” in *Research in Computational Molecular Biology: 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10–13, 2020, Proceedings*. Springer Nature, p. 104.

List of Figures

List of Tables