Chapter 2

The FM-Index and Genome Assembly

2.1 Introduction

2.1.1 Publication Note

The work described in this chapter was previously published in [Simpson and Durbin, 2010]. Section 2.6 describes unpublished results. The work described is the sole work of the author, under the supervision of his PhD supervisor, Richard Durbin.

2.2 Definitions and Notation

Let X be a string of symbols $a_1, ..., a_l$ from an alphabet Σ. The length of X is denoted $|X|$. $X[i] = a_i$ is the i-th symbol of X and $X[i, j]$ is the substring $a_i, ..., a_j$. Let $X' = a_l, a_{l-i}, ..., a_1$ denote the reverse of X. When discussing lexicographically ordered alphabets, $b+1$ will refer to the next highest symbol in the alphabet after b.

If Y is a substring of X and $Y \neq X$, then we say that Y is a *proper substring* of X. A substring $X[k, |X|]$ is a *suffix* of X and a substring $X[1, k]$ is a *prefix* of

X. We will often refer to substrings of length n as a n-mers [1].

When discussing text indices, we will consider all strings to be terminated by a sentinel symbol \$ that is not in Σ and is lexicographically lower than all the symbols in Σ. In this work the DNA alphabet will be used. The lexicographic ordering of this alphabet and the sentinel is $\$ < A < C < G < T$.

2.2.1 Genomes and Sequence Reads

We define a *genome* to be a long string from the alphabet $\{A, C, G, T\}$ representing the complete DNA sequence of an individual, for simplicity ignoring potential subdivisions into chromosomes. A sequence *read* is a short substring from a genome. DNA is a double stranded molecule and sequence reads can originate from either strand. We use the notation $\overline{X}$ for the *reverse-complement* of a read X.

Reads may contain sequencing errors. These occur when the sequencing instrument incorrectly identifies a symbol (substitution error), or when a symbol is incorrectly inserted into, or deleted from, the string (indel error). We will occasionally assume that reads are error-free. It will be clearly stated when this is the case.

We say that two reads X and Y *overlap* if a prefix of X is equal to a suffix of Y or vice versa. If X and Y originate from opposite strands, they overlap if the reverse complement of one of them overlaps the other. If X (or $\overline{X}$) has the same sequence as Y then we say that the two reads are *identical*, or *duplicates*. If X (or $\overline{X}$) is a proper substring of Y, then we say that X is *contained* within Y. When two reads X and Y overlap, we can merge them into a new sequence Z which contains both X and Y. In this case we say that we have *assembled* X and Y. We will refer to the new sequences that result from assembly as *contigs*.

In a shotgun sequencing experiment a set of sequence reads is randomly sampled from a genome, G, with an unknown sequence. We will denote the indexed set of reads by $\mathcal{R}$, with the i-th read in the set denoted by $\mathcal{R}_i$. The *de novo* assembly problem is to reconstruct the sequence of G given only $\mathcal{R}$. If the sequence of

[1] Such substrings are also referred to as n-grams or n-tuples, particularly in Computer Science. We will use n-mer for consistency with most literature in the sequence assembly field.

G was drawn randomly from $\{A, C, G, T\}$ the assembly problem would be easy - even very short reads with length on the order of $\log |G|$ would suffice to assemble nearly all of G unambiguously. In reality, the problem is far more complicated. Eukaryotic genomes are shaped by the duplication and divergence of large segments, along with the proliferation of transposon elements. The difficulty of the assembly problem stems from these *repetitive* regions.

2.3 Assembly Graphs

To help reconstruct G from $\mathcal{R}$, we can build a graph of the relationships between sequence reads. We will discuss three different types of assembly graph - the overlap graph, the de Bruijn graph and the string graph. The common thread between these graphs is that the structure of the underlying genome is reflected in the structure of the graph. Walks through these graphs describe assemblies of the reads into segments of the genome. We begin with the overlap graph.

2.3.1 Overlap Graphs

In the overlap graph each sequence read in $\mathcal{R}$ is a vertex. Two vertices are joined by an edge if their corresponding reads overlap. To help distinguish true overlaps from spurious overlaps we set a threshold of τ_{min} on the minimum acceptable overlap length. When allowing for sequencing errors, a threshold on the maximum error rate of ϵ_{max} will also be set. For the remainder of this chapter, we will consider only error-free reads. This constraint will be relaxed in Chapter 3. We associate coordinates with each edge describing the matching segments of the linked reads.

The overlap graph is computationally demanding to construct. A naive algorithm (suitable only for very small sequencing projects) would compare all pairs of reads to discover overlapping pairs. Such an algorithm has $O(N^2)$ time complexity where $N = \sum_{i=1}^{|\mathcal{R}|} |\mathcal{R}_i|$. For larger sequencing projects, comparing all pairs of reads is impractical. To accelerate overlap detection, an index of all l-mer sequences appearing in the reads can first be constructed, and only reads sharing

an l-mer would be checked for an overlap, avoiding the need to compare all pairs. In Myers [2005] the use of a q-gram filter (subsequently described in Rasmussen et al. [2006]) is suggested for finding all (τ_{min}, ϵ_{max}) overlaps in $O(N^2/D)$ time where D is a function of the amount of memory available. In Gusfield [1997] an algorithm to solve the all-pairs maximal overlap problem in $O(N + |\mathcal{R}|^2)$ time using a suffix tree is described. The quadratic term is due to the requirement that an overlap between all pairs must be found - if we instead require that only τ_{min}-overlaps are found, a faster version of this algorithm is possible. However, the suffix tree requires a very large amount of memory to store [Abouelhoda et al., 2004], so in practice this data structure is not commonly used for indexing large sequence collections.

An optimal algorithm for overlap detection would require $O(N + |E|)$ time, where $|E|$ is the number edges in the resulting graph. Even with such an algorithm, overlap-based assemblers suffer from two computational problems. First, as all reads covering the same position of the genome will mutually overlap, the number of overlaps (and therefore edges in the graph) is quadratic in sequencing depth. This is a significant problem when assembling high-throughput sequence data as the genomes tend to be covered very deeply to ensure each base is covered multiple times (typically greater than 40 reads cover each base). Second, for reads originating from repetitive regions, the number of overlaps will be quadratic in the product of the sequencing depth and the number of copies of the repeat. This leads to greatly increased computational cost and a much larger graph when assembling highly repetitive genomes. For this reason, overlap assemblers occasionally take the step of masking known repeats and low-complexity sequence as a pre-processing step, as exemplified by Celera's assembly of the human genome [Venter et al., 2001].

As each read in $\mathcal{R}$ was sampled from a distinct location in G, the optimal solution will assign a linear ordering $\{1, 2, ..., |\mathcal{R}|\}$ to the elements of $\mathcal{R}$, reflecting their position along G. Such a solution requires finding a path through the overlap graph which visits each vertex exactly once. This is the Hamiltonian path problem which is known to be NP-complete. For this reason a global solution to the assembly problem is rarely sought. Instead, the assembly of reads into *contigs* typically focuses on finding local groups of reads that can be unambiguously

assembled together.

2.3.2 de Bruijn Graphs

We follow Pevzner's formulation of the de Bruijn graph [Pevzner et al., 2001]. Set a fixed value ρ and let $\mathcal{P}$ be the set of all distinct ρ-mers in $\mathcal{R}$. Let $k = \rho - 1$ and $\mathcal{V}$ be the set of all distinct k-mers in $\mathcal{R}$. $\mathcal{V}$ is the set of vertices of the graph. For each $P \in \mathcal{P}$ we create a bidirected edge $K_1 \leftrightarrow K_2$ where K_1 is the length-k prefix of P and K_2 is the length-k suffix of P. To handle the double-stranded nature of DNA, we can also introduce the reverse-complements of the ρ-mers and k-mers into the graph[1].

Unlike the overlap graph, the de Bruijn graph is computationally easy to construct. The construction of the vertex and edge set only requires iterating over distinct k or p-mers, which can easily be implemented with hash tables, sorted arrays of strings, red-black trees or any other data structure allowing efficient queries for whether a string is present in a collection. A second important property, perhaps the most important, is how repeats appear in the graph. Let R be a long repeated substring of G ($|R| > k$). By definition each instance of R contains the same sequence of k-mers. As the de Bruijn graph only contains *distinct* k-mers, there is a single vertex for each of these k-mers. Therefore unlike the overlap graph the repeat copies do not contribute extra edges to the graph - all copies are represented by a single segment of the graph (see figure 1.3). Coupled with the efficient construction algorithms, this property has made the de Bruijn graph the dominant data structure for assembly of genomes from high-throughput short read data.

The reconstruction of G from the de Bruijn graph requires a tour that visits each edge at least once. This is the route-inspection problem (also known as the Chinese Postman Problem). Pevzner proposed [2001] to introduce edge multiplicities in the graph to transform the problem into one in which each edge must be visited exactly once. This is a classic graph theory problem originally studied by Euler [Euler, 1741] and hence known as a Eulerian path problem.

[1] Some assemblers, like ABySS, represent a k-mer and its reverse complement as a single vertex

As the Eulerian path problem has a known polynomial-time algorithm [Fleury, 1883], this was a promising approach to reducing the computational complexity of genome assembly. However, with long repeats in the genome the problem is underconstrained - many possible solutions may exist with only one representing the true sequence of G [Nagarajan and Pop, 2009]. For this reason, assembling contigs from the de Bruijn graph mainly focuses on local segments of the graph that can be unambiguously assembled, like in the case of the overlap graph.

2.3.3 The String Graph

As a refinement to the overlap graph, Myers formulated the String Graph [2005]. The String Graph has a number of important differences with the overlap graph. First, we remove duplicated or contained reads from $\mathcal{R}$ to provide a new non-redundant vertex set. Second, we label each edge with the unmatched substrings of each read. Let $X = X_1X_2$ and $Y = Y_1Y_2$ be two overlapping reads. When X and Y are from the same sequencing strand, either $X_2 = Y_1$ or $X_1 = Y_2$. When X and Y are from opposite sequencing strands, either $\overline{X_1} = Y_1$ or $\overline{X_2} = Y_2$. We will call the substrings that are found in both X and Y the *matched* substrings. The other substrings are the *unmatched* substrings. We define the labels of an edge to be the unmatched substrings of the reads. Specifically:

$$L_{xy} = \begin{cases} Y_2 & \text{if } X_2 = Y_1 \\ Y_1 & \text{if } X_1 = Y_2 \\ \overline{Y_2} & \text{if } X_1 = \overline{Y_1} \\ \overline{Y_1} & \text{if } X_2 = \overline{Y_2} \end{cases}$$

The reciprocal label L_{yx} is defined similarly. Note that in the case that X and Y are from opposite sequencing strands, then the label is the reverse-complement of the unmatched substring. The concatenation of X and L_{xy} is an assembly of reads X and Y - the resulting string contains both the sequence of X and Y. As there are no duplicated or contained reads in the graph, the labels are necessarily non-empty strings.

We can also associate with each edge in the graph a *type* describing the relationship between the pair of reads it links.

$$type_{xy} = \begin{cases} S & \text{if } X_2 = Y_1 \text{ or } X_2 = \overline{Y_2} \\ P & \text{if } X_1 = Y_2 \text{ or } X_1 = \overline{Y_1} \end{cases}$$

If a suffix of X overlaps a prefix of Y, we will call the edge an SP-edge. Likewise when a prefix of X overlaps a suffix of Y, we will call the edge a PS-edge. When a reverse-complemented prefix (suffix) of X overlaps a prefix (suffix) of Y, we call the edge a PP-edge (SS-edge). We note that when $type_{xy} = type_{yx}$ X and Y are necessarily from opposite sequencing strands.

Walks through the graph must respect the edge types. For example, for the walk $X \leftrightarrow Y \leftrightarrow Z$ to be valid, if $type_{yx} = S$ then $type_{yz}$ must be P and vice versa. In other words, if we enter a vertex via a suffix overlap, we must leave the vertex using a prefix overlap. This makes the string graph *bidirected*. We can associate a string with a walk through the graph by concatenating the label of each edge in the walk to the sequence of the first vertex in the walk.

Definition 1. *Let $X_1 \leftrightarrow X_2 \leftrightarrow ... \leftrightarrow X_n$ be a valid walk through the edge-labelled graph. We assume that $X_1, X_2, ..., X_n$ are from the same sequencing strand. If not, we preprocess the walk by changing the strand of each edge label to match the strand of X_1. After such a step, we define the string corresponding to the walk to be:*

$$A_{x_1 x_2 ... x_n} = \begin{cases} X_1 L_{x_1 x_2} ... L_{x_{n-1} x_n} & \text{if } type_{x_1 x_2} = S \\ L_{x_{n-1} x_n} ... L_{x_1 x_2} X_1 & \text{if } type_{x_1 x_2} = P \end{cases}$$

The final difference between the string graph and the overlap graph is that *transitive* edges are removed.

Definition 2. *Consider a read X that overlaps reads Y and Z, which mutually overlap. The initial overlap graph will contain the edges $X \leftrightarrow Y$, $X \leftrightarrow Z$ and $Y \leftrightarrow Z$. We will say an edge $X \leftrightarrow Z$ is* transitive *when the string spelled by path $X \leftrightarrow Z$ is the same as the string spelled by path $X \leftrightarrow Y \leftrightarrow Z$.*

Transitive edges can be removed from the graph without reducing the set of strings that can be spelled by the graph. We will refer to non-transitive edges as *irreducible*. We will now define useful properties of transitive edges. For

the following properties we will assume without loss of generality that $type_{xy} = type_{xz} = S$ and all three reads are from the same strand. This implies $X \leftrightarrow Y$, $X \leftrightarrow Z$ and $Y \leftrightarrow Z$ are all SP-edges.

Property 1. *The label of the transitive edge $X \leftrightarrow Z$ is the concatenation of the edge labels of the walk $X \leftrightarrow Y \leftrightarrow Z$.*

Property 2. *L_{xy} is a prefix of L_{xz}.*

Proof. From the definition of a transitive edge $A_{xz} = A_{xyz}$. From definition 1, $A_{xz} = XL_{xz}$ and $A_{xyz} = XL_{xy}L_{yz}$ therefore $L_{xz} = L_{xy}L_{yz}$ and L_{xy} is a prefix of the transitive edge.

□

Property 3. *Let C be the matched substring between X and Z. C is also a substring of Y.*

Proof. We can write X and Z in terms of C as $X = X'C$ and $Z = CL_{xz}$. Assume C is not a substring of Y and let $C = C_1C_2$ where C_2 is the prefix of Y that matches a suffix of X and C_1 is not empty. Write Y and Z in terms of these substrings as $Y = C_2L_{xy}$ and $Z = C_1C_2L_{xz}$. From property 1 we have $L_{xz} = L_{xy}L_{yz}$ therefore $Z = C_1C_2L_{xy}L_{yz}$. This implies that Y is contained within Z which contradicts a precondition on the graph therefore C must be a substring of Y.

□

Property 4. *C is not a prefix of Y*

Proof. The proof is similar to that of property 3 except we assume the prefix of Y is C to arrive at a contradiction.

□

Property 5. *The overlap between X and Y is longer than the overlap between X and Z.*

Proof. Again let C be the matched substring between X and Z. The length of C is the length of the overlap between X and Z. As C is a substring of Y but

not a prefix of Y, the matched substring between X and Y is MC where M is non-empty. The length of MC is therefore greater than the length of C.

□

An example string graph built from three overlapping reads is given in figure 2.1.

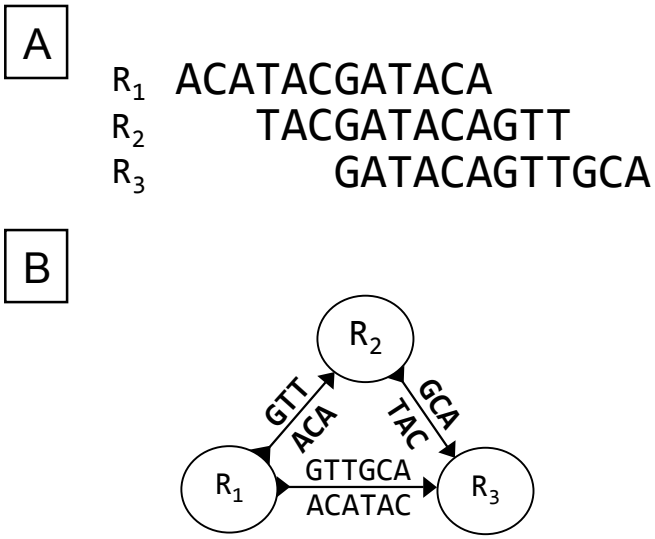

Figure 2.1: Diagram of a simple assembly graph. Three overlapping reads (R_1, R_2, R_3) are shown in panel A. Panel B shows the graph constructed from the overlaps between the reads. The arrowheads pointing into the nodes depict an edge of type P and arrowheads pointing away from the nodes depict edges of type S. For example the edge between R_1 and R_2 is a SP-edge. The edge $R_1 \leftrightarrow R_3$ is transitive. Removing this edge will turn the graph into a string graph.

Transforming an overlap graph into a string graph by removing duplicated and contained reads, along with transitive edges, avoids the quadratic expansion of edges with sequencing depth and repeat copy number. Like in the de Bruijn graph, repeats are collapsed to single segments in the graph. The string graph therefore represents an alternative to the de Bruijn graph, with the important benefit that the graph contains the full read sequences, representing the complete information present in $\mathcal{R}$.

The string graph can be built indirectly by first constructing an overlap graph then removing duplicate and contained reads, then removing transitive edges. Myers provides an $O(|E|)$ expected-time algorithm to perform transitive reduction on an existing overlap graph Myers [2005]. The fundamental problem of overlap assembly remains however, in that the computation of the overlap graph is the computational bottleneck. This is the problem that we address in this chapter by devising an algorithm to *directly* output the string graph, without the need to transitively reduce an overlap graph. This algorithm will allow us to construct the graph in linear-time, bringing the algorithmic complexity of the string graph in line of that of the de Bruijn graph. We begin the description of this algorithm with an introduction to text indices.

2.4 The Suffix Array, BWT and FM-Index

The *suffix array* data structure was introduced by Manber and Myers [1990] as a succinct representation of the lexicographic ordering of the suffixes of a string. The suffix array of a string X, denoted $\mathbf{SA}_X$, is a permutation of the integers $\{1, 2, ..., |X|\}$ such that $\mathbf{SA}_X[i] = j$ iff $X[j, |X|]$ is the i-th lexicographically lowest suffix of X. For example, if $X = \text{AAGTA\$}$ then $\mathbf{SA}_X = [6, 5, 1, 2, 3, 4]$. Since the suffix array is a sorted data structure, the start positions of all the instances of a pattern Q in X will occur in an interval in $\mathbf{SA}_X$. We refer to such an interval as a *suffix array interval* and associate with it a pair of integers $[l, u]$ denoting the first and last index in $\mathbf{SA}_X$ that correspond to a position in X of an instance of Q. Using $\mathbf{SA}_X$ and the original string X, l and u can be efficiently found with a binary search for Q. Ferragina and Manzini [2000] developed a related method of indexing text, called the FM-index, which requires considerably less memory than a suffix array and can compute l and u in $O(|Q|)$ time, independent of the size of the text being searched. Central to the FM-index is the Burrows-Wheeler transform (BWT). Originally developed for text compression [Burrows and Wheeler, 1994] the Burrows-Wheeler transform of X, denoted $\mathbf{B}_X$, is a permutation of the symbols of X such that:

$$\mathbf{B}_{\mathrm{X}}[i] = \begin{cases} X[\mathbf{SA}_{\mathrm{X}}[i] - 1] & \text{if } \mathbf{SA}_{\mathrm{X}}[i] > 1 \\ \$ & \text{if } \mathbf{SA}_{\mathrm{X}}[i] = 1 \end{cases}$$

Restated, $\mathbf{B}_{\mathrm{X}}[i]$ is the symbol preceding the first symbol of the suffix starting at position $\mathbf{SA}_{\mathrm{X}}[i]$. For the example string X from above, $\mathbf{B}_{\mathrm{X}} = AT\$AAG$.

Ferragina and Manzini extended the BWT representation of a string by adding two additional data structures to create a structure known as the FM-index. Let $\mathbf{C}_{\mathrm{X}}(a)$ be the index in $\mathbf{SA}_{\mathrm{X}}$ of the first suffix starting with symbol a. If v is the number of symbols lexicographically lower than a in X, then $\mathbf{C}_{\mathrm{X}}(a) = v + 1$. Let $\mathbf{Occ}_{\mathrm{X}}(a, i)$ be the number of occurrences of the symbol a in $\mathbf{B}_{\mathrm{X}}[1, i]$[1]. We note that $\mathbf{C}_{\mathrm{X}}$ and $\mathbf{Occ}_{\mathrm{X}}$ include counts for the sentinel symbol, \$.

$\mathbf{C}_{\mathrm{X}}(a)$ for the example string X is:

a	\$	A	C	G	T
$\mathbf{C}_{\mathrm{X}}(a)$	1	2	5	5	6

$\mathbf{Occ}_{\mathrm{X}}(a, i)$ for X is:

a	\$	A	C	G	T
$\mathbf{Occ}_{\mathrm{X}}(a, 1)$	0	1	0	0	0
$\mathbf{Occ}_{\mathrm{X}}(a, 2)$	0	1	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 3)$	1	1	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 4)$	1	2	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 5)$	1	3	0	0	1
$\mathbf{Occ}_{\mathrm{X}}(a, 6)$	1	3	0	1	1

Using $\mathbf{C}_{\mathrm{X}}(a)$ and $\mathbf{Occ}_{\mathrm{X}}(a, 1)$, Ferragina and Manzini provided an algorithm to search for a string Q in X. Let S be a string whose suffix array interval is

[1]These definitions use 1-based coordinates. When implementing these data structures 0-based coordinates are preferred. To allow this, we modify the definition of $\mathbf{C}_{\mathrm{X}}(a)$ to equal v and $\mathbf{Occ}_{\mathrm{X}}(a, i)$ to count over $\mathbf{B}_{\mathrm{X}}[0, i]$. The following algorithms work in either case.

known to be $[l, u]$. The interval for the string aS can be calculated from $[l, u]$ using $\mathbf{C}_{\mathrm{X}}$ and $\mathbf{Occ}_{\mathrm{X}}$ by the following:

$$l' = \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, l-1) \tag{2.1}$$

$$u' = \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, u) - 1 \tag{2.2}$$

We encapsulate equations (2.1) and (2.2) in the following algorithm, `updateBackward`.

Algorithm 1 updateBackward($[l, u]$, a)

$l \leftarrow \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, l-1)$
$u \leftarrow \mathbf{C}_{\mathrm{X}}(a) + \mathbf{Occ}_{\mathrm{X}}(a, u) - 1$
return $[l, u]$

To search for a string Q, we need to first calculate the interval for the last symbol in Q then use equations (2.1) and (2.2) to iteratively calculate the interval for the remainder of Q. The initial interval for a single symbol a is simply $[\mathbf{C}_{\mathrm{X}}(a), \mathbf{C}_{\mathrm{X}}(a{+}1){-}1]$ where $a{+}1$ denotes the next largest symbol in the alphabet[1]. The `backwardsSearch` algorithm presents the searching procedure in detail. If `backwardsSearch` returns an interval where $l > u$, Q is not contained in X otherwise $\mathbf{SA}_{\mathrm{X}}[i]$ is the position in X of each occurrence of Q for $l \leq i \leq u$.

Algorithm 2 backwardsSearch(Q) - find the interval in $\mathbf{SA}_{\mathrm{X}}$ for the pattern Q

$i \leftarrow |Q|$
$l \leftarrow \mathbf{C}_{\mathrm{X}}(Q[i])$
$u \leftarrow \mathbf{C}_{\mathrm{X}}(Q[i]+1) - 1$
$i \leftarrow i - 1$
while $l \leq u$ & $i \geq 1$ **do**
  $[l, u] \leftarrow$ `updateBackward`$([l, u], Q[i])$
  $i \leftarrow i - 1$
return $[l, u]$

The `backwardsSearch` algorithm requires updating the suffix array interval $|Q|$ times. As each update is a constant-time operation, the complexity of `backwardsSearch` is $O(|Q|)$ given that the FM-index is already constructed.

[1] If a is the largest symbol in Σ, then $\mathbf{C}_{\mathrm{X}}(a+1)$ simply returns $n+1$ where n is the highest index in $\mathbf{SA}_{\mathrm{X}}$

2.4.1 The Generalized Suffix Array

We can easily expand the definition of a suffix array to include sets of strings. Let $\mathcal{T}$ be an indexed set of strings and $\mathcal{T}_i$ be element $\mathcal{T}[i]$. We define $\mathbf{SA}_\mathcal{T}[i] = (j, k)$ iff $\mathcal{T}_j[k, |\mathcal{T}_j|]$ is the i-th lowest suffix in $\mathcal{T}$. In the generalized suffix array, unlike the suffix array of a single string, two suffixes can be lexicographically equal. We break ties in this case by comparing the indices of the strings. In other words we treat each string in $\mathcal{T}$ as if it was terminated by a unique sentinel character $\$_i$ where $\$_i < \$_j$ when $i < j$. We extend the definition of the Burrows-Wheeler transform to collections of strings as follows. Let $\mathbf{SA}_\mathcal{T}[i] = (j, k)$ then:

$$\mathbf{B}_\mathcal{T}[i] = \begin{cases} \mathcal{T}_j[k-1] & \text{if } k > 1 \\ \$ & \text{if } k = 1 \end{cases}$$

Like the BWT of a single string, $\mathbf{B}_\mathcal{T}$ is a permutation of the symbols in $\mathcal{T}$; therefore the definitions of the auxiliary data structures for the FM-index, $\mathbf{C}_\mathcal{T}(a)$ and $\mathbf{Occ}_\mathcal{T}(a, i)$, do not change.

2.5 Direct Construction of the String Graph

In this section we describe the first results of this work, string graph construction algorithms based on the FM-index of a set of reads. We will show that by using the FM-index of $\mathcal{R}$ the set of overlaps can be computed in $O(N + C)$ time for error-free reads where C is the total number of overlaps found. We then provide an algorithm which detects only the overlaps for irreducible edges - removing the need for the transitive reduction algorithm and allowing the direct construction of the string graph.

2.5.1 Building an FM-index from a set of sequence reads

To build the FM-index of $\mathcal{R}$, we can first compute the generalized suffix array of $\mathcal{R}$. We could do this by creating a string which is the concatenation of all members of $\mathcal{R}$, $S = \mathcal{R}_1\mathcal{R}_2...\mathcal{R}_m$ and then use one of the well-known efficient suffix array construction algorithms to compute $\mathbf{SA}_\mathrm{S}$ [Puglisi et al., 2007]. We have adopted a different strategy and have modified the induced-copying suffix array

construction algorithm [Nong et al., 2009] to handle an indexed set of strings $\mathcal{R}$ where each suffix array entry is a pair (j, k) as described in section 2.4.1. This suffix array construction algorithm is similar to the Ko-Aluru algorithm [2005]. A set of substrings of the text (termed LMS substrings) is sorted from which the ordering of all the suffixes in the text is induced. Our algorithm differs from the Nong-Zhang-Chan algorithm as we directly sort the LMS substrings using multikey quicksort [Bentley and Sedgewick, 1997] instead of sorting them recursively. This method of construction is fast in practice as typically only $30 - 40\%$ of the substrings must be directly sorted. Once $\mathbf{SA}_{\mathcal{R}}$ has been constructed, the Burrows-Wheeler transform of $\mathcal{R}$, and hence the FM-Index is easily computed as described above. We also compute the FM-index for the set of *reversed* reads, denoted $\mathcal{R}'$, which is necessary to compute overlaps between reverse complemented reads. We also output the *lexicographic index* of $\mathcal{R}$, which is a permutation of the indices $\{1, 2, ..., |\mathcal{R}|\}$ of $\mathcal{R}$ sorted by the lexicographic order of the strings. This can be found directly from $\mathbf{SA}_{\mathcal{R}}$ and is used to determine the identities of the reads in $\mathcal{R}$ from the suffix array interval positions once an overlap has been found.

Alternatively, when all reads in $\mathcal{R}$ are short ($\approx$ 100bp) then the Bauer-Cox-Rosone algorithm [Bauer et al., 2011] can be used to construct $\mathbf{B}_{\mathcal{R}}$. This topic will be revisited in 3.2.1 when discussing our software implementation.

2.5.2 Overlap detection using the FM-Index

We now consider the problem of computing the set of τ_{min} overlaps between reads in $\mathcal{R}$. Consider two reads X and Y. If a suffix of X matches a prefix of Y a SP-edge will be created in the initial overlap graph. We will describe a procedure to detect overlaps of this type from the FM-index of $\mathcal{R}$. Let X be an arbitrary read in $\mathcal{R}$. If we perform the `backwardsSearch` procedure on the string X, after k steps we have calculated the interval $[l, u]$ for the suffix of length k of X. The reads indicated by the suffix array entries in $[l, u]$ therefore have a substring that matches a suffix of X. Our task is to determine which of these substrings are prefixes of the reads. Recall that if a given element in the suffix array, $\mathbf{SA}_{\mathcal{R}}[i]$, is a prefix of a string then $\mathbf{SA}_{\mathcal{R}}[i] = (j, 1)$ for some j and $\mathbf{B}_{\mathcal{R}}[i] = \$$ by definition. Therefore, if we know the suffix array interval for a string Q, the interval for the

strings beginning with Q can be determined by calculating the interval for the string $\$Q$ using equations (2.1) and (2.2). This interval, denoted $[l_\$, u_\$]$, indicates that the reads with prefix Q are the $l_\$$-th to $u_\$$-th lexicographically lowest strings in $\mathcal{R}$. We can therefore recover the indices in $\mathcal{R}$ of the reads overlapping X using lexicographic index of $\mathcal{R}$. The algorithm is presented below in `findOverlaps`.

Algorithm 3 findOverlaps(X, τ) - determine the reads in $\mathcal{R}$ that overlap X by at least τ symbols

```
i ← |X|
l ← C_R(X[i])
u ← C_R(X[i] + 1) − 1
i ← i − 1
while l ≤ u & i ≥ 1 do
  if |X| − i + 1 ≥ τ then
    [l_$, u_$] ← updateBackwards([l, u], $)
    if l_$ ≤ u_$ then
      outputOverlaps(X, [l_$, u_$])
  [l, u] ← updateBackward([l, u], X[i])
  i ← i − 1
if l ≤ u then
  outputContained(X, [l, u])
```

The `findOverlaps` algorithm is similar to the backwards search procedure presented in section 2.4. It begins by initializing $[l, u]$ to the interval containing all suffixes that begin with the last symbol of X. The interval $[l, u]$ is then iteratively updated for longer suffixes of X. When the length of the suffix is at least the minimum overlap size, τ, we determine the interval for the reads that have a prefix matching the suffix of X and output an overlap record for each entry (using the subroutine `outputOverlaps`). When the update loop terminates, $[l, u]$ holds the interval corresponding to the full length of X. The `outputContained` procedure writes a containment record for X if X is contained by any read in $[l, u]$. The overlaps detected by `findOverlaps` correspond to SP-edges. We must also calculate the overlaps for SS-edges and PP-edges, which arise from overlapping reads originating from opposite strands. To calculate SS-edges we use `findOverlaps` on the complement of X (not reversed) and the FM-index of $\mathcal{R}'$. Similarly, to calculate PP-edges we use `findOverlaps` on $\overline{X}$ (the reverse

complement of X) and the FM-index of $\mathcal{R}$.

In rare cases, multiple valid overlaps may occur between a pair of reads. In this case the interval set returned by `findOverlaps` will contain intersecting intervals. To account for this, we sort the intervals and only keep the interval representing a maximal overlap when two adjacent intervals intersect.

The overlap records created by `outputOverlaps` are constructed in constant time as they only require a lookup in the lexicographic index of $\mathcal{R}$. Let c_i be the number of overlaps for read $\mathcal{R}_i$. The `findOverlaps` algorithm makes at most $|\mathcal{R}_i|$ calls to `updateBackwards` and a total of c_i iterations in `outputOverlaps` for a total complexity of $O(|\mathcal{R}_i| + c_i)$. For the entire set $\mathcal{R}$, the complexity is $O(N + C)$ where $C = \sum_{i=1}^{|\mathcal{R}|} c_i$. Note that the majority of these edges are transitive and subsequently removed. We can therefore improve this algorithm by only outputting the set of irreducible edges, allowing the direct construction of the string graph. We address this in the next section.

2.5.3 Detecting irreducible overlaps

To directly construct the string graph, we must only output irreducible edges. Recall from section 2.3.3 that the labels of the irreducible edges for a given read are prefixes of the labels of transitive edges. We use this fact to differentiate between irreducible and transitive edges during the overlap computation. Consider a read X and the set of reads that overlap a suffix of X, $\mathcal{O}$. We could devise an algorithm to find the subset consisting only of irreducible edges by calculating the edge-labels of all members of $\mathcal{O}$ and filtering out the members whose label is the extension of the label of some other read. This would require iterating over all members of $\mathcal{O}$ which can be quite large for repetitive reads or high-depth data. We will now show that the labels of the irreducible edges can be constructed directly from the suffix array intervals using the FM-index.

Consider a substring S that occurs in $\mathcal{R}$ and its suffix array interval $[l, u]$. Let a *left extension* of S be a string of length $|S| + 1$ of the form aS. We can use $\mathbf{B}_{\mathcal{R}}[l, u]$ to determine the set of left extensions of S. Let $\mathcal{B}$ be the set of symbols that appear in the substring $\mathbf{B}_{\mathcal{R}}[l, u]$. The left extensions of S are

the strings aS such that $a \in \mathcal{B}$. Note that we do not have to iterate over the range $\mathbf{B}_\mathcal{R}[l, u]$ to determine $\mathcal{B}$. Since $\mathbf{Occ}_\mathcal{R}(a, i)$ is defined to be the number of times symbol a occurs in $\mathbf{B}_\mathcal{R}[1, i]$ we can count the number of occurrences of a in $\mathbf{B}_\mathcal{R}[l, u]$ (and hence aS in $\mathcal{R}$) in constant time by taking the difference $\mathbf{Occ}_\mathcal{R}(a, u) - \mathbf{Occ}_\mathcal{R}(a, l-1)$. If the \$ symbol occurs in $\mathbf{B}_\mathcal{R}[l, u]$ we say that S is *left terminal*, in other words one of the elements of $\mathcal{R}$ has S as a prefix. We similarly define a *right extension* of S as a string of length $|S|+1$ of the form Sa. While we cannot build the right extensions of S directly from the FM-index, the right extensions of S are equivalent to left extensions of S' (the reverse of S) in $\mathcal{R}'$. Let S be *right terminal* if \$ exists in $\mathbf{B}_{\mathcal{R}'}[l', u']$, in other words S is a suffix of some string in $\mathcal{R}$.

The procedure to find all the irreducible edges of a read X and construct their labels is to find all the intervals containing the prefixes of reads that overlap a suffix of X, then iteratively extend them rightwards until a right-terminal extension is found. The terminated read forms an irreducible edge with X and the label of the edge is the sequence of bases that were used during the right-extension. All non-terminated strings with the same sequence of extensions are transitive and therefore not considered further.

The algorithm requires searching the FM-index in two directions, first backwards to determine the intervals of overlapping prefixes and then forwards to extend those prefixes and build the irreducible labels. Naively this would require first determining the intervals $[l, u]$ for each matching prefix, P, and then reversing the prefix and performing a backwards search on the FM-index of $\mathcal{R}'$ to find the interval $[l', u']$ for P'. The intervals $[l', u']$ would then be used in the extension stage to determine the labels of the irreducible edges. We can do better however by noting that the interval $[l', u']$ can be calculated directly during the backwards search without using the FM-index of $\mathcal{R}'$. We define $\mathbf{OccLT}_\mathcal{R}(a, i)$ to be the number of symbols that are lexicographically lower than a in $\mathbf{B}_\mathcal{R}[1, i]$. Let $S = X[i, |X|]$ be a suffix of X and $[l_i, u_i]$ its suffix array interval. Suppose we know the interval $[l'_i, u'_i]$ for S' in $\mathcal{R}'$. Let $a = X[i-1]$. The interval for $S'a = [l'_{i-1}, u'_{i-1}]$ is therefore:

$$l'_{i-1} = l'_i + (\mathbf{OccLT}_\mathcal{R}(a, u_i) - \mathbf{OccLT}_\mathcal{R}(a, l_i - 1)) \tag{2.3}$$

$$u'_{i-1} = l'_{i-1} + (\mathbf{Occ}_{\mathcal{R}}(a, u_i) - \mathbf{Occ}_{\mathcal{R}}(a, l_i - 1) - 1) \tag{2.4}$$

The interval for $X'[1]$ is identical to that of $X[|X|]$ since $\mathbf{B}_{\mathcal{R}}$ and $\mathbf{B}_{\mathcal{R}'}$ are both permutations of symbols in $\mathcal{R}$ therefore $\mathbf{C}_{\mathcal{R}} = \mathbf{C}_{\mathcal{R}'}$. We can therefore initialize the interval $[l', u']$ to the same initial value of $[l, u]$ and perform a forward search of X' simultaneously while performing a backward search of X using only the FM-index of $\mathcal{R}$. This does not require any additional storage as the $\mathbf{OccLT}_{\mathcal{R}}$ array can easily be computed from $\mathbf{Occ}_{\mathcal{R}}$ by summing the values for symbols less than a. This procedure is similar to the 2way-BWT search recently proposed by Lam *et al.* (2009) . The `updateFwdBwd` algorithm implements equations (2.3) and (2.4) along with `updateBackward` to calculate the pair of intervals. The $\mathcal{F}$ parameter to `updateFwdBwd` indicates the FM-index used - that of $\mathcal{R}$ or $\mathcal{R}'$.

Algorithm 4 updateFwdBwd($[l, u, l', u']$, a, $\mathcal{F}$)

$l' \leftarrow l' + (\mathbf{OccLT}_{\mathcal{F}}(a, u) - \mathbf{OccLT}_{\mathcal{F}}(a, l - 1))$
$u' \leftarrow l' + (\mathbf{Occ}_{\mathcal{F}}(a, u) - \mathbf{Occ}_{\mathcal{F}}(a, l - 1) - 1)$
$[l, u] \leftarrow$ `updateBackwards`$(l, u, a, \mathcal{F})$
return $[l, u, l', u']$

We now give the full algorithm for detecting the irreducible overlaps for a read X. The algorithm is performed in two stages, first a backwards search on X is performed to collect the set of interval pairs, denoted $\mathcal{I}$, for prefixes that match a suffix of X. This algorithm is presented in `findIntervals` below and is conceptually similar to `findOverlaps`.

The interval set found by `findIntervals` is processed by `extractIrreducible` to find the intervals corresponding to the irreducible edges of X. This algorithm has two parts. First, the set of intervals is tested to see if some read in the interval set is right terminal. If so, the intervals corresponding to the right terminal reads form irreducible edges with X and are returned. If no interval has terminated, we create a subset of intervals for each right extension of $\mathcal{I}$ and recursively call `extractIrreducible` on each subset.

The algorithm above assumes that $\mathcal{R}$ does not have any contained reads. If this is not the case, a slight modification must be made. If the set of reads overlapping X includes a read that is a proper substring of some other read it is possible that

Algorithm 5 findIntervals(X, τ)

$\mathcal{I} \leftarrow \emptyset$
$i \leftarrow |X|$
$l \leftarrow C(X[i])$
$u \leftarrow C(X[i]+1)-1$
$[l', u'] \leftarrow [l, u]$
$i \leftarrow i-1$
while $l \leq u$ & $i \geq 1$ **do**
  if $|X| - i + 1 \geq \tau$ **then**
    $[l_\$, u_\$, l'_\$, u'_\$] \leftarrow \texttt{updateFwdBwd}([l, u, l', u'], \$, \mathcal{R})$
    if $l_\$ \leq u_\$$ **then**
      $\mathcal{I} \leftarrow \mathcal{I} \cup [l_\$, u_\$, l'_\$, u'_\$]$
  $[l, u, l', u'] \leftarrow \texttt{updateFwdBwd}([l, u, l', u'], X[i], \mathcal{R})$
  $i \leftarrow i-1$
return $\mathcal{I}$

Algorithm 6 extractIrreducible($\mathcal{I}$)

if $\mathcal{I} = \emptyset$ **then**
  return $\emptyset$
$\mathcal{L} \leftarrow \emptyset$
for all $[l, u, l', u'] \in \mathcal{I}$ **do**
  $[l'_\$, u'_\$, l_\$, u_\$] \leftarrow \texttt{updateFwdBwd}([l', u', l, u], \$, \mathcal{R}')$
  if $l_\$ \leq u_\$$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \cup [l_\$, u_\$]$
if $\mathcal{L} \neq \emptyset$ **then**
  return $\mathcal{L}$
for all $a \in \Sigma$ **do**
  $\mathcal{I}_a \leftarrow \emptyset$
  for all $[l, u, l', u'] \in \mathcal{I}$ **do**
    $[l'_a, u'_a, l_a, u_a] \leftarrow \texttt{updateFwdBwd}([l', u', l, u], a, \mathcal{R}')$
    if $l_a \leq u_a$ **then**
      $\mathcal{I}_a \leftarrow \mathcal{I}_a \cup [l_a, u_a, l'_a, u'_a]$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \texttt{extractIrreducible}(\mathcal{I}_a)$
return $\mathcal{L}$

the first right terminal extension found is not that of an irreducible edge but of the contained read. It is straightforward to handle this case by observing that such a read will have an overlap that is strictly shorter than that of the irreducible edge. In other words, the only acceptable right terminal extension is to the reads in $\mathcal{I}$ that have the longest overlap with X.

We can similarly modify `extractIrreducible` to handle overlaps for reads from opposite strands. To do this, we use `findIntervals` to determine the intervals for overlaps for the same strand as X and overlaps from the opposite strand of X (using the complement of X as in the previous section). When extending an interval that was found by the complement of X, we extend it by the complement of a. In other words if we are extending same-strand intervals by A, we extend opposite strand intervals by T and so on.

We now offer a sketch of the complexity of the irreducible overlap algorithm in the case where all edges in the graph are part of the walk spelling the genome sequence G. Let L_i be the label of irreducible edge i. During the construction of L_i at most k_i intervals must be updated, corresponding to the number of reads that have an edge-label containing L_i. The sum over all irreducible edges, $E = \sum_i (|L_i|k_i)$, is the total number of interval updates performed by `extractIrreducible`. Note that each read in $\mathcal{R}$ is represented by a path through the string graph. The total number of times edge i is used in the set of paths spelling all the reads in $\mathcal{R}$ is k_i and the amount of sequence in $\mathcal{R}$ contributed by edge i is $|L_i|k_i$. This implies E can be no larger than N, the total amount of sequence in $\mathcal{R}$, and `extractIrreducible` is $O(N)$. As `findIntervals` is also $O(N)$, the entire irreducible overlap detection algorithm is $O(N)$.

2.5.4 Results

The algorithms described in this chapter form the basis of the assembler I developed, SGA[1]. As a proof of concept, I profiled these algorithms on simulated error-free sequence reads. The assembly is broken into three stages: index, overlap and assemble. The index stage constructs the FM-index for a set of sequence reads, the overlap stage computes the set of overlaps between the reads and

[1]String Graph Assembler

the assemble stage loads the graph, performs transitive reduction if necessary, then compacts unambiguous paths in the graph and writes them out as a set of contigs. I performed two sets of simulations. In all simulations the faster Bauer-Cox-Rosone algorithm was used to calculate the FM-index. In both sets of simulations, I compared the exhaustive overlap algorithm (which constructs the set of all overlaps) and the direct construction algorithm (which only outputs overlaps for irreducible edges). First, I simulated *E. coli* reads with average sequence depth from 5X to 100X to investigate the computational complexity of the overlap algorithms as a function of sequence depth. After constructing the index for each data set, I ran the overlap step in exhaustive and direct mode with fixed $\tau = 27$. The running times of these simulations are shown in figure 2.2. As expected, the direct overlap algorithm scales linearly with sequence depth. The exhaustive overlap algorithm exhibits the expected above-linear scaling as the number of overlaps for a given read grows quadratically with sequence depth.

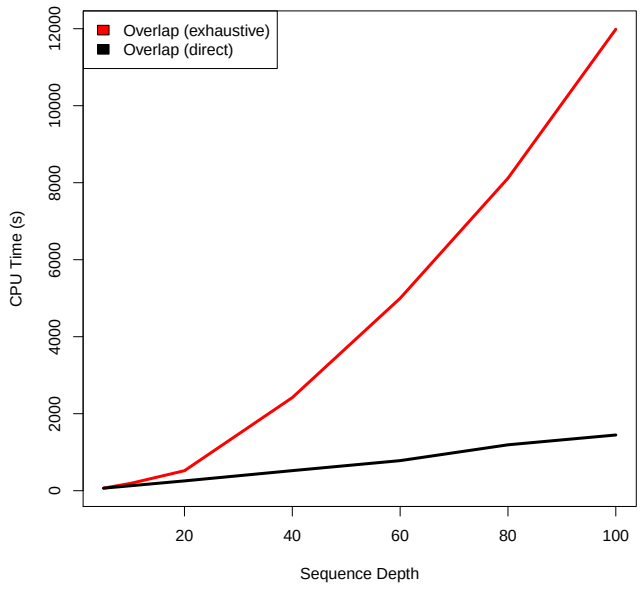

Figure 2.2: The running time of the direct and exhaustive overlap algorithms for simulated E. coli data with sequence depth from 5X to 100X.

I also simulated data from human chromosomes 22, 15, 7 and 2 to assess how the algorithms scale with the size of the genome. I pre-processed the chromosome sequences to remove sequence gaps then generated 100bp error-free reads randomly at an average coverage of 20X for each chromosome. Again I compared the direct construction algorithm to the exhaustive construction algorithm. The overlap length was set to 45. The results of these simulations are summarized in table 2.1.

	chr 22	chr 15	chr 7	chr 2	ratio
Chr. size (bp)	34.9M	81.7M	155.4M	238.2M	6.8
Number of reads	7.0M	16.3M	31.1M	47.6M	6.8
Duplicated reads	684k	1,663k	3,103k	4,709k	6.9
Duplicated %	9.8%	10.2%	10.0%	9.9%	-
Transitive edges	70.0M	176.4M	364.2M	583.8M	8.3
Irreducible edges	7.2M	17.2M	36.2M	57.4M	8.0
Assembly N50 (bp)	3.0k	4.1k	4.3k	4.8k	-
Longest contig (bp)	41.4k	51.9k	63.2k	57.9k	-
Index time	1,486s	3,652s	7,284s	11,443s	7.7
Overlap time (e)	3,595s	9,393s	27,736s	30,176s	8.4
Overlap time (d)	2,204s	7,885s	11,516s	17,596s	8.0
Assemble time (e)	1,399s	4,795s	15,287s	33,140s	23.7
Assemble time (d)	280s	694s	1,518s	2,432s	8.7
Index memory	1.0GB	2.2GB	4.2GB	6.5GB	6.5
Overlap mem. (e)	0.5GB	1.2GB	2.3GB	3.5GB	7.0
Overlap mem. (d)	0.5GB	1.2GB	2.3GB	3.5GB	7.0
Assemble mem. (e)	8.9GB	24.5GB	27.2GB	99.7GB	11.2
Assemble mem. (d)	2.1GB	5.0GB	10.0GB	15.7GB	7.5

Table 2.1: Simulation results for human chromosomes 22, 15, 7 and 2. For the overlap and assemble rows, (e) and (d) indicate the exhaustive and direct algorithms, respectively. The last column is the ratio between chromosome 2 and 22.

For all chromosomes the direct overlap computation algorithm was faster. The direct overlap calculation step required almost half the run time when com-

pared to the exhaustive overlap calculation. When the full overlap graph was constructed (exhaustive case) the assemble step required performing Myers' transitive reduction algorithm on the graph. This step was over 23 times longer for chromosome 2 than chromosome 22, as the chromosome 22 graph contained over 500 million transitive edges that needed to be removed. The vast number of transitive edges in this case caused the chromosome 2 graph to require nearly 100GB of memory. The assemble step for the direct case was over 13 times faster on chromosome 2 as the initial graph was much smaller and transitive reduction did not need to be performed. These results verify the efficiency of my direct string graph construction algorithm and the benefits when compared to building the full overlap graph first.

The memory bottleneck in these assemblies is loading the string graph into memory during the assemble step. This bottleneck will be addressed in the next chapter, where I describe an algorithm to find and compress *unipaths* in the graph without the need to load the entire string graph in memory.

2.6 Representing a de Bruijn Graph using the FM-Index

Recall from section 2.3.2 that the set of distinct k-mers of $\mathcal{R}$ defines the vertices of its de Bruijn graph. Likewise, the set of distinct ρ-mers defines the edges of the graph. This observation allows us to use the FM-index as a representation of the de Bruijn graph of $\mathcal{R}$. Here we describe queries to compute the local structure of the graph around a single vertex. To test whether a given k-mer is a vertex in the graph, we can use algorithm `isDBGVertex`. This performs a simple backwards search to check whether the k-mer, or its reverse-complement, has a non-empty suffix array interval.

Algorithm 7 isDBGVertex(K, $\mathcal{R}$)

$[l_1, u_1] \leftarrow$ `backwardsSearch`$(K, \mathcal{R})$
$[l_2, u_2] \leftarrow$ `backwardsSearch`$(\overline{K}, \mathcal{R})$
if $l_1 \leq u_1$ or $l_2 \leq u_2$ **then**
  return true
else
  return false

We can also use the FM-Index to get the neighbors of a particular vertex in the graph. For a k-mer K, there are 8 possible neighbors. To find which are actually present in $\mathcal{R}$, we can simply directly query for the 16 possible ρ-mers describing these neighbors (including their reverse complements). Pseudocode for this algorithm is shown in `getDBGNeighborsSingleIndex`. This requires $16(k+1)$ interval updates.

If the FM-index of both $\mathcal{R}$ and $\mathcal{R}'$ is available, we can implement a faster procedure based on performing extension queries like those described in our string graph construction algorithm in section 2.5.3. We start by calculating the interval pair for K, then using $\mathbf{Occ}_{\mathcal{R}}$ to calculate the possible left-extensions of K. This query provides the ρ-mers of the form xK which define prefix neighbors of K. To calculate suffix neighbors of K (of the form Kx), we can use right-extension queries. These queries must also be performed with the reverse complement of K, to cover both strands. In total, this procedure requires $4K$ interval updates plus 8 accesses of the occurrence array. As we need the FM-index of $\mathcal{R}$ and $\mathcal{R}'$ to do the right-extension queries, the memory usage is doubled when compared to `getDBGNeighborsSingleIndex`.

The description within this section uses Pevzner's ρ-mer based formulation of the de Bruijn graph. In our implementation of these algorithms we use a slight modification. Instead of querying for ρ-mers, we directly query for the neighboring k-mers (of the form $xK[1, k-1]$ and $K[2, k]x$ for $x \in \{a, c, g, t\}$). This is subtly different as connected vertices do not necessarily need to share a ρ-mer - they only need to overlap by $k-1$ bases.

Algorithm 8 getDBGNeighborsSingleIndex(K, $\mathcal{R}$)

$k \leftarrow |K|$
for all $Q \in \{aK, cK, gK, tK\}$ **do**
  $[l, u] \leftarrow$ `backwardsSearch`$(Q, \mathcal{R})$
  $[l', u'] \leftarrow$ `backwardsSearch`$(\overline{Q}, \mathcal{R})$
  if $l \leq u$ or $l' \leq u'$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{Q[1, k]\}$
for all $Q \in \{Ka, Kc, Kg, Kt\}$ **do**
  $[l, u] \leftarrow$ `backwardsSearch`$(Q, \mathcal{R})$
  $[l', u'] \leftarrow$ `backwardsSearch`$(\overline{Q}, \mathcal{R})$
  if $l \leq u$ or $l' \leq u'$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{Q[2, k+1]\}$
return $\mathcal{I}$