Chapter 1

Introduction

1.1 Genomic Variation

Genomic variation is one of the fundamental aspects of biology. Difference in the DNA-sequence between two individuals can lead to a change in the translated RNA sequence and further in the sequence, and thereby the form and function, of the expressed protein. Or it can lead to less drastic changes such as changes in the RNA-structure or the shape of the DNA molecule itself. In this way genomic variation can determine differences between specimens of the same species and also differences between the species themselves.

Within a species, the genomic variation between individuals are often limited by evolutionary conservative pressure, meaning that the difference in DNA-sequence between viable specimens of the same species are often small and does not lead to big changes in neither the structure of the DNA or the translated proteins. This limitation has made possible the use of *reference genome*s for a species, a DNA-sequence though to represent the generic sequence for that species, where individual variation from the reference sequence are though to be small.

Reference genomes have helped make sense of sequencing data, that have been dominated by large numbers of small sequence fragments. Without a good reference genome, one would need to fit all those sequence fragments together like a puzzle, in a process called *assembly*. Using a reference genome, one can instead find the best sequence match for each read in the reference sequence, and thus find both where each read belongs in the genome, and also how the sequence differs from the reference. Finding the position of each read can give context to them, since the location of biologically important parts of the genome can be represented on the genome. For instance, if the sequencing of an individual maps to a location within a the known location of a protein coding gene and differs from the reference sequence, it's possible that that individual has a genomic variant that alters the function of that protein.

Since this process of *mapping* sequence reads to the reference is such a

fundamental step in many biological analyses, the quality of the reference sequence has been of high importance. Thus, since the dawn of human genome sequencing in *year*, the Genome Reference Consortium has released n versions of the human reference genome alone. The latest version of the human reference genome (GRCh38), highlights some shortcomings of representing the reference of a species as a single sequence. GRCh38 includes, in addition to the traditional linear reference sequence, a set of alternative sequences from areas of the genome where there are known variations of the DNA-sequences which makes it problematic to map reads from those areas to the reference sequences. Secondly the new version included changes that disrupted the coordiante system of the reference. This has lead to a backward incompatibility that has prevented a widespread adoption of the new reference.

1.2 Mapping Bias

Mapping sequencing reads to the reference entails finding subsequences in the reference sequence that are highly similar to the sequenced reads. The match can be inexact due to either the actual DNA-sequence being different or sequencing errors that can substitute one nucleotide with another. It is necessary to set a limit to how dissimilar the reference subsequence can be in order to produce a match due to computational complexity and that allowing a high level of divergence can lead to a large number of matches. For instance, *BWA-mem* by default requires a shared subsequence of at least 19 bp in order to produce a match. This means that reads from regions with much variation from the reference can be unmappable using standard mapping software, and be susceptible to small amount of sequenceing error yielding them unmappable. This means that the resemblence of the sample to the reference genome will lead to better mapping quality, and that some regions of the genome will have a lower mapping rate than others.

1.3 Geometry of the reference genome

As well as being an indexable lookup table for sequencing reads, a reference genome provides a coordinate system and a geometry for sequence data that allows us to look at different sequence elements in conjunction. Most important is the analysis of overlap and distances between subsequences. For instance, the location of a potential transcription factor binding site, predicted from a ChIP-seq experiment, can be compared to the positions of known genes, predicted from amongst others RNA-seq experiments, to determine which gene is most likely regulated by the binding of a TF to the binding site. The distances between subsequences is also used in some mapping tools themselves, as a way to evaluate the match of a read to a reference subsequence. For instance Minimap2 [**?**] uses the relative positioning of kmer matches to the reference to find which chain of kmer-anchors match the read sequence best. The distance be-

tween two subsequences on the reference is invariant to SNPs, since they do not affect distances. However indels and especially structural variants affect the distances between subsequences, and can thus change the outcome of any analysis involving distances. For instance, a structural variant that changes which gene a regulatory element affects.

1.4 Sequence Graphs

A DNA-sequence can be represented by a sequence of letters from the alpahbet (A, C, G, T). In nature it is common for DNA-sequences to be highly similar to each other, only separated by small variations caused by mutations. The most common such mutations are single base-pair mutations, commonly called single-nucleotide-polymorphism (SNP), and insertions and deletions of short subsequences. The most common format for representing similiar DNA-sequences are a block format where deletions are represented by a 'blank' symbol '_'. This format has a redundancy in the representation of the shared parts of the sequences. This works fine for small sets of short sequences, but for sequences on the genomic scale the redundancy gets significant, and the number of '-' symbols needed gets two big. Sequence graphs are an alternative to the block format for representing multiple similar sequences. In its basic form a sequence graph is a collection of nodes, representing nucleotides, and a set of edges representing neighbouring pairs of nucleotides. A single nucleotide sequence can then be represented as the nucleotides of the sequence connected by edges. Similarily two sequences that are separated by a single SNP can be represented as such... Any set of sequences represented in block format can be uniquely represented by a sequence graph. In addition to the graph, a representation of which paths each sequence takes through the graph is needed in order to contain the same information as the block format. Even without thes specific paths, the graph representation of the sequences contain meaningful information. They succinclty sum up the variation between the sequences. Also any path through the sequence graph is a possible combination of the sequences present that can represent a sequence obtained by recombination events from the available sequences. Also each path through the graph represents a possible common ancestor from the graph.

On a larger scale the block format is unconvenient due to the redundancy in the sequences. It is then more common to use a single sequence as a reference sequence and then represent the different sequences only by the places they are different from the reference. This is common for example when representing mulitple genomes from the same species. The variant common format? (VCF) format is a common such format that represents the. It contains a list of variants represented by a position in the reference sequence, a subsequence from the refernce from that positions, and the alternative sequence that replaces the reference sequence. In addition it can contain extra columns for known haplotypes, specifying which variants are present in each haplotype.

Such vcf representations can also be represented uniquely by a sequence

graph, by first representing the reference sequence as a linear sequence graph and then adding to the graph the nodes and edges to represent the variants. It is also here necessary to represent the haplotypes as paths through the graph in order to keep the haplotype information.

The most common variations (SNPs and indels), leads to directed acyclic graphs (DAG) when converted to a sequence graphs. However other types of variants does not have this property and thus leads to more complicated graphs.

Other types of variants Large scale variants that affects the ordering of the nucleotides are not well suited to being represented by DAGs. An example of this is transpositions, where a subsequence of DNA is moved to another location in the genome. This can be represented as a DAG by adding a new variant in the graph covering the whole sequence from covering both the old an new postions of the subsequence. However this can lead to much redundancy since the sequence between the two positions will be represented twice. The other alternative is to only add new edges to the graph, representing the sequence when the substring has been moved. This will however break the acyclisity of the graph, and thus complicate most operations one would do on the graph.

Another case which will lead to redundancy if represented as a graph is reversals. Here a piece of the DNA is reversed leading to the subsequence being substituded with its reverse complement. Adding a subsequence in the graph representing the reverse compliment is inderictly redundant. Even though the reverse compliment is not included as a path in the graph, it is directly deducible from a sequence in the graph. In order to represent suche reversals, and other things needing the reverse compement, the concept of a side graph has been introduced. In a side graph, all the nodes representing nucleotides have to sides and an edge is defined as going from one side of a node to a side of another node. One node-side represents the nucleotide, while the other side represents the complement in the other reading direction.

Applications Multiple sequence alignment An early application of sequence graphs was in sequence alignment. The application made a sequence graph of the pairwise alignments and also made it possible to align a sequence graph to another sequence graph. A central theme in this application was that if an an alignment of two sequences minimizes some edit distance between them, then any path through the sequence graph representing this alignment will represent a possible common ancestor of the two sequences that minimizes the combined edit distance to the two sequences. Thus in an iterative pairwise joining scheme, one can in each step achieve a sequence graph that represents the most likely common ancestor and then align these sequence graphs to each other. Here, the graph representation is clearly intuitive and benefitial. The fact that each possible path through the graph represents something meaningful makes the graph format very succinct and benefitial for this application.

Mapping In recent years, mapping reads to such sequence graphs have gathered much attention. The process of mapping reads to a reference sequence is trying to find out where in the reference sequence a read is from and usually involves an index that can quickly look up subsequences from the reference. Common such indexes for linear reference sequences are the FM-index, that uses

the burrows wheeler transform to look up subsequences in linear time, and kmer based indexes that can look up subsequences of constant length in constant time. Both of these types of indexes encounters problems when applied to a graph, due to the combinatorial growth of possible subsequences. The GCSA2 index is a kmer-based index able to index a generic sequence graph, but needs to prune out variants in complex regions in order to restrain the combinatiorial growth in kmers. It also requires significantly more computaional time and memory to create the index than for linear references. Another problem with mapping to a graph based references is that combining several subsequence matches is also hard. The early attempts at graph-mapping had the goal of finding any path in the graph. It is however doubful that this is the right approach due to two facts. First when including variants from many individuals in the graph, a region can be filled up with variants from many different samples where none of them appear in the same individual. Thus a region of the graph can include very many paths, where just a marginal fraction of them actually represents sequences that are seen in the samples or could be attained by a small set of recombinations. This is problematic since sequences will have an increased chance of mapping to such areas. The mismapping in itself is an issue, but this will also lead to a bias toward such regions, so any downstream analysis of the data might be severly compromised. A possible remedy for this is to only search in sequences represented by a real haplotype, that is using the haplotype path information in the mapping. This avoids the problem both of combinatorial growth of subsequences and also that of mapping sinks. It is however a question of whether the graph format is meaningful in this context, as the mapping is in reality linear.

Downstream analysis The possiblity to map reads to a variation graph leads to possibilities for downstream analysis. The most natural is to call variants. The process of variant calling is using mapped reads to determine in which parts of the sequence of the sample differs from the sequence in the reference. In this case, graph mapped reads can be advantageous.

1.5 Formalism

pWe define a *simple sequence graph* $SG = \{G, s\}$ over an alphabet Γ is a graph $G = \{V, E\}$ and a label function $s : V \to \Gamma$ labeling each vertex with a letter from Γ. The set of all sequences in SG is

$$\Sigma(SG) = \{s(v_1), s(v_2), , , s(v_n) s.t. v_i \in V \wedge (v_i, v_{i+1}) \in E\}$$

Similarily a *compacted sequence graph* $CG = \{G, s\}$ over alphabet Γ is a graph G and label function $s : V \to \Sigma^*/\epsilon$ labelling each vertex to a non-empty finite string over Γ. The set of all sequences in SG is

$$\Sigma(CG) = \Sigma_0(CG) \cup \Sigma_1(CG)$$
$$\Sigma_0(CG) = \{s(v)_{a:b} s.t. v \in V \wedge 0 < a < b \leq |s(v)|\}$$
$$\Sigma_1(CG) = \{(s(v_1)_{a:}, s(v_2), , , s(v_n)_{:b}) s.t. v_i \in V \wedge (v_i, v_{i+1}) \in E \wedge a > 0 \wedge b \leq |v_n|\}$$

For DNA-sequence graphs over the alphabet $\Gamma = \{A, C, G, T\}$ and the reverse compliment function $rc : \Gamma \to \Gamma$, we get all sequences reverse sequences by

$$\Sigma_{rc}(SG) = \{(rc(s_n), rc(s_{n-1}), , , rc(s_1)) \forall s = (s_1, s_2, , , s_n) \in \Sigma(SG)\}.$$

and the set of all possible sequences on either forward or reverse strand is given by $\Sigma_{all}(SG) = \Sigma(SG) \cup \Sigma_{rc}(SG)$.

In order to represent reversals succinctly one needs a generalization of the sequence graph as given in [**?**]. A *reversible sequence graph* is given by $RG = \{V, E\}$, where $E \subseteq (V \times \{-1, 1\})^2$. Here the possible sequences in the graph is represented by:

$$\Sigma(RG) = \{s(v_1, d_1), s(v_2, d_2), , , s(v_n, d_n) s.t. v_i \in V \wedge ((v_i, d_i), (v_{i+1}, d_{i+1})) \in E\}$$

$$s(v, d) = \begin{cases} s(v_1), & \text{if} d = 1 \\ rc(s(v_1), & \text{if} d = -1 \end{cases}$$

1.6 The Importance for Non-Redundancy

The simplicity of working with acyclic sequence graphs compared to generalized sequcence graphs with possible cycles are big, both in terms of algorithmic effiency and inituitvity. Beacuse of this a compelling argument needs to be made for dealing with generalized sequence graphs. Here we will disucss the benefit of these more complex structuers, both in terms of mapping and interpreting intervals and locations on the graphs.

1.6.1 Mapping

A key element of the read mapping procedure is to filter out reads that map with similar scores to multiple places in the reference. Repeats in the reference of the same subsequences is thus problematic, since often a read mapping to the subsequence, will have mathces in many of the repeats. The result of this is that the read will get a low mapQ score. This will however lead to a loss of information, since a read mapping to a reapeated subsequence can still give information about where in the genome it belongs, and it will also be possible to see if there is a variation in the read from the references subsequence. In this case, representing the repeated subsequences as a cycle in the graph is benefitial, since as much information as possible about the read mapping is kept. A similar case can be made for representing reversals in generalized sequence graphs. In a simple graph, the reversed sequence would need duplicated representation, yielding two matches for each read. These multiple mapping issues could also be alieviated by including information in the mapping algorithm that specifies which regions are reversals and duplications of each other, akin to the alt-allele handling in BWA-mem. However representing the location of the resulting alignments would be problematic, which leads to the next benefit of genralized graph structures: representation.

1.6.2 Representing locations and intervals

The output from the read mapping is generally the location on the reference of the match, and a description of how the read sequence diverges from the reference sequences at that location. The location of the mappings can be used in downstream analysis. It is then beneficial if the location of the read gives as much information about the whereabout of the mapping as possible. Without the possibility of cycles, reapeats and reversals would need to be represented as either the set of locations representing the sequences, or a random location from this set.

1.7 Data Structures and Algorithms

The focus of this thesis is on the representation and handling of intervals and positions in compressed sequence graphs. The nonlinearity of sequence graphs means that the complexity of interval representation and handling becomes an issue. In a linear genome both representation and handling of intervals can usually be done in constant complexity. An interval can be represented as simply a start and end position along with a specifier of direction (s, e, d). Similarily distances between two in intervals can be computed in constant time by

$$D(I_1, I_2) = \max(0, s_1 - e_2, s_2 - e_2)$$

and the overlap between two intervals by

$$|I_1 \cap I_2| = \max(\min(e_1, e_2) - \max(s_1, s_2), 0)$$

. Paper 1 discussed the implication of breaking this inherent simplicty when having a graphical and not linear reference.

In the general case a position on the graph needs to be represented by a tuple specifying the id of the node and the offset, and an interval needs a start position, an end position and the specification of each node covered by the interval. This means that the memory cost of representing an interval grows linearly with the interval length. Also, the distance between two positions is not uniquely defined, but depends on which path through the graph is taken between the two points. And finding specific distances, such as the shortest or longest possible distance, does not in the general case have constant complexity, but rather depends on both the distance itself, and the complexity of the graph.

Also off particular importance to Graph Peak Caller was the operation of finding all positions within a distance d from a position P. On a linear coordinate system, this is imply represented by the range $[P, P + d)$. However on a graph this constitutes finding each position reachable by a path of length $< d$ starting on P. Using a breadth first search, this can be on average obtained in $O(kd)$ time.

Graph Peak Caller suffered performance wise from these additional complexities. Where MACS2 can call peaks on n reads in the time frame of a couple of

minutes on a standard laptop, GPC used close to a day on several CPUs and used more memory than available on a standard laptop.

GPC2 introduces some simplifications to the graph structure that allowed it to run much closer to MACS2 running times. Going away from *vg*s data structure of nodes of maximum length of often 32 and storing each variant as a separate node. It is worth to note that a single SNP in that format increases the number of nodes by 3 the number of edges by 4. Coupled with the fact that the vast majority of variants are SNPs, this means that the majority of edges in the graph represents variants that does not affect the distance. It was therefore possible to reduce the number of edges, and thereby the complexity of the graph k, by representing SNPs in a separate structure.

Throughout the project two open source python libraries were developed and optimized, in order to work efficiently with sequence graphs. *Offsetbased Graph* provides basic functionality while *Graph Peak Caller* provides methods necessary for doing peak calling on sequence graphs. The final implementations and algorithms is described below.

1.7.1 Sequence Graph

The main data structure provided is a full graph, having as members a *Graph*, describing the topology of the sequence graph, a *Sequences* object, giving access to the sequence on each node and a *Path* object, describing the path the linear reference takes through the graph.

A *Graph* object represents a graph $G = (V, E)$ and a node labeling $L : V \rightarrow \mathbb{N}$ where the vertex set is assumed to be a dense set of integers starting at 1, $V = \{1, 2, , , N\}$, and the labeling L describes the sequence length of each node. The Graph object also contains a representation of the positions and sequence of all SNPs in a SNP object. The vertex set V and vertex labeling L is represented as an array of length N where each element is the sequence length of the corresponding vertex. The set of edges E is described by a ragged array adj_list equivalent of an adjacency list so that adj_list[i]=$\{v : v \in V \wedge (i, v) \in E\}$. For convenience the graph object also contains a reverse adjacency list such that: adj_list[i]=$\{v : v \in V \wedge (v, i) \in E\}$. The implementation of the ragged arrays are described later in this section. The SNP class is a ragged array having as elements the location of all SNPs on all vertices. Such that $SNP[i] = \{(p, i) :\}$ *describe SNP set.* The Sequences class is a ragged array where each element is the sequence of that node. Sequences[i]=s(i). The Path class contains an array node_ids of length n_p which describes which nodes the path traverses, and an array distance_to_node of length $n_p + 1$ giving the distance along the reference path to each node in the path, such that (distance_to_node[i], distance_to_node[i+1]) gives the interval vertex i represents along the reference path.

1.7.2 Creating from Variants

We start with a reference sequence $G \in \Gamma^*$ and a set of variants described in the following.

We let $S \subset \mathbb{Z}_{<|G|} \times \Gamma$ be a set of SNPs where (i, c) represents a SNP at position i to letter c. A set of insertions $I \subset \mathbb{Z}_{<|G|} \times \Gamma^*$ where $(i, s) \in I$ represents an insertion at position i with sequence s. A set of deletions $D \subset \mathbb{Z}_{<|G|} \times \mathbb{N}^+$ where $(i, l) \in D$ represents an deletion at position i with length l.

From a variation set $V = (S, I, D)$ we create the graph by the following method. Breakpoints $B = \{i \mid \exists s\,[(i, s) \in I]\} \cup \{i \mid \exists l\,[(i, l) \in D \vee (i - l, l) \in D]\}$. Let $(b_1, b_2, ..., b_{n_r+2})$ be the ordered elements of $B \cup \{0, |G|\}$. We then get a reference graph as

$$\begin{aligned} G_r &= (V_r, E_r) \\ V_r &= \{1, 2, , , n_r\} \\ E_r &= \{(i, i+1) | i \in \{1, 2, , , n_r - 1\}\} \\ s(i) &= G[b_i : b_{i+1}] \end{aligned}$$

The set of deletion edges is $E_d = \{(i, j) | (b_i, (b_j - b_i)) \in D\}$. Letting $I_s = (i_1, i_2, , , i_{|I|})$ be the sorted elements of I, the set of insertion vertices and edges are given by:

$$\begin{aligned} V_i &= \{n_r + 1, n_r + 2, , , n_r + |I|\} \\ E_{-1} &= \{(r - 1, k) \mid k \in \{1, 2, , , |I|\} \wedge b_r = i_k\} \\ E_1 &= \{(k, r) \mid k \in \{1, 2, , , |I|\} \wedge b_r = i_k +\} \\ E_i &= E_{-1} \cup E_1 \end{aligned}$$

Properly describe insertions. The full graph can then be made by $G = (V_r \cup V_i, E_r \cup E_i \cup E_d)$. SNPs is then represented such that $SNP_i = \{(p - b_i, c) \mid (p, c) \in S \wedge b_i \leq p < b_{i+1}\}$. *vertex labels*

1.7.3 Peak Calling

Graph Peak Caller is based on the peak caller *MACS2*, which algorithm is briefly explained below. The input is a set of intervals $\{(s_k, e_k, d_k)\}$. The first step is to count how many extended reads cover each position of the reference genome, where each interval is extended to a length equaling a previously estimated fragment length f. I.e for each position i we want to find the count

$$P(i) = |\{k \mid d_k = 1 \wedge s_k \leq i < s_k + f\} \cup \{k \mid d_k = -1 \wedge e_k - f \leq i < e_k\}|$$

The pileup function P is represented sparsely by a set of indices $\{s_k\}$ and values $\{v_k\}$ such that

$$\forall k \forall j \in \{s_k, s_k + 1\}, , , s_{k+1} (P(j) = v_k)$$

the indices and values are found algorithmically as

```
def count_extended(intervals, f):
    starts = [s if d==1 else e-f for s, e, d in intervals]
    ends = [e if d==1 else s+f for s, e, d in intervals]
    changes = starts+ends
    args = argsort(changes)
    codes = [1 if arg<=len(starts) else -1 for arg in args]
    s = changes[args]
    v = cumsum(codes)
    return s, v
```

Which is done in $O(n \log n)$ time. The counts for each position is compared with a background track which represents a local average of reads. This is generated by using a large extension length and dividing the pileup values by the extension size. s, v = count_extended(control_intervals, E)/E; v/=E. Assuming that each count $P(i)$ are Poisson distributed as P_i $Poisson(\lambda_i)$, a null hypothesis of $H_0^i : \lambda_i = C(i), H_a^i : \lambda_i > C(i)$ is made for each position and a p-value calcluated: $p_i = P(P_i >= P(i))$ *too many ps.* Too adjust for multiple testing, a final set of q values is calculated as $q_i = p_i N_i$ where $N_i = |\{k \in \{1, 2, , |G|\} \mid p_k \leq p_i\}|$. The q values are thresholded on a given significance level α so we get $T(i) = q_i \leq \alpha$.

```
def get_p_values(pileup, background):
    indices = pileup.indices + background.indices
    indices.sort()
    pileup_values = pileup.values[search_sorted(indices, pileup.indices)]
    background_values = background.values[search_sorted(indices,
        background.indices)]
    p_values = poisson.sf(pileup_values, background_values)
    return Pileup(indices, p_values)
```

Which is done in $O(n)$ time.

```
def get_q_values(p_values):
    counts = diff(p_values.indices)
    args = argsort(p_values.values)
    values, idxs = unique(p_values, return_index=True)
    N = cumsum(counts[args])[idxs]
    q_values = N*values
    all_q_values = zeros_like(p_values.values)
    all_q_values[idxs] = diff(q_values)
    all_q_values = cumsum(all_q_values)
    restored = zeros_like(all_q_values)
    restored[args] = all_q_values
    return Pileup(p_values.indices, restored)
```

The thresholded values are obtained simply by Pileup(q_values.indices, q_values.values>alpha).

The final peaks are called in two steps from the thresholded values. First, joining peak intervals that are separated by less than the estimated read length, and secondly removing peaks that are shorter than the estimated fragment length f. Both steps can be made using the same function.

```
def remove_small(indices, size):
    indices = indices.reshape((-1, 2))
    lengths = indices[:, 1]-indices[:, 0]
    big_enough = lengths>=size
    return indices[big_enough].ravel()

peaks = r_[indices[0], remove_small(indices[1:-1], read_length),
    indices[-1]]
peaks = remove_small(peaks, fragment_length)
```

1.7.4 Graph Peak Caller

The following is a brief description of the algorithms and data structures used in the adatptation of MACS2 to graph genomes. It is assumed here that the sequence graph is a compressed simple sequence graph, and therefore also acyclic. We also assume that the node ids are sorted according to a topological sort of the graph.

The fundamental data structure in for graph peak caller is the linear representation that converts coordinates from $LR : \mathbb{N}_{|V|} \times \mathbb{Z} \to \mathbb{N}_N$, where $N = \sum_{i=0}^{|V|-1} L(i)$ is the number of positions in the graph. This linear mapping is represented in the LinearRepr class, which has an array node_offsets where node_offsets[i]=$\sum_{k=0}^{i} L(k)$ such that a coordinate $(v, o) \in \Gamma$ is converted to $i \in \epsilon$ by node_offsets[v]+o. The inverse conversion of a coordinate $i \in \epsilon$ is done by v=searchsorted(node_offset, i); o = i−node_offsets[v])

A pileup is created in the linear coordinate space by the following method. For each interval (s, e, v, d) we define the extended interval as a tuple

$$ext(I_k, f) = \begin{cases} (\{s_k\}, E_k^+, X_k^+), & \text{if } d = 1 \\ X_k^-, E_k^-, \{e_k\}), & \text{if } d = -1 \end{cases}$$

where

$$E_k^+ = \{(v_1, v_2) \in E \mid D(e_k, (v_2, 0)) \leq f\} \cup \{(v_1, v_2), (v_2, v_3), , , (v_{k-1}, v_k)\}$$
$$X_k^+ = \{((v, o) \in \Gamma \mid D(e_k, (v, o)) = f\}$$
$$E_k^+ = \{(v_1, v_2) \in E \mid D((v_2, 0), s_k, \leq) f\} \cup \{(v_1, v_2), (v_2, v_3), , , (v_{k-1}, v_k)\}$$
$$X_k^+ = \{((v, o) \in \Gamma \mid D((v, o), s_k) = f\}$$

From this we can get the sparse pileup as

$$\begin{aligned}
\delta(LP((v,o))) &= N_s - N_e + D_E \\
N_s &= |\{(X_s, E, X_e) \in Ext \mid (v,o) \in X_s\}| \\
N_e &= |\{(X_s, E, X_e) \in Ext \mid (v,o) \in X_s\}| \\
D_E &= |\{(X_s, E, X_e) \in Ext \mid \exists v_0 \in V\,[(v_0, v) \in E]\}| \\
&\quad - |\{(X_s, E, X_e) \in Ext \mid \exists v_2 \in V\,[(v, v_2) \in E]\}|
\end{aligned}$$

Chapter 2

Summary of Papers

In paper 1, we discussed the implications of using a graphical reference structure for representing and comparing genomic locations and intervals. The work focused on how to obtain succinct and interpretable representations that were robust against changes to the reference graph topology. Paper 2 developed a new method of calling ChIP-seq peaks using a sequence graph as reference. The method is an adaptation of MACS2 that uses the benefits of read-mapping to a variation graph to obtain more accurate peak calling of potential transctription factor binding sites. Paper 3 further developed the method of paper 2 to perform better on graphs constructed solely from SNPs and indels from a VCF-file. Introducing a new succinct data structure for variation graphs and obtaining similar speeds to MACS2. Paper 4 looked into the potential for using a two step approach for read mapping using the benefits of mapping to a graph for estimating a genotype for the sample, before using a linear mapper to obtain a final mapping that is not suffer from the too big search space of graph mapping.