Chapter 4

Algorithms for Variant Detection from an Assembly Graph

4.1 Introduction

In the preceding chapters, I described algorithms for assembling the complete genome of an individual from a set of sequence reads. Often we are only interested in the ways in which two (or more) genomes differ. For example, we may be interested in the differences between a sequenced individual and the reference genome for a species, or differences between a child and its parents. I will refer to the problem of finding genomic differences between closely related genomes as *variant detection*.

Reference-based variant detection algorithms map and align reads to a reference genome then find substrings of the reads that are consistently different with respect to the reference. As reads contain sequencing errors, the differences between the reads and reference are typically assessed using a probabilistic model to help distinguish between errors (either due to base-calling errors during sequencing or mis-aligning the reads to the reference) and true variants. While mapping-based algorithms have become the standard method for variant detection, it is considerably more difficult to accurately find indel variants than it is to find substitution variants [Li and Homer, 2010]. For this reason, assembly-based variant calling algorithms have been proposed [Catchen et al., 2011; Iqbal et al., 2012]. In chapter 3 we saw how allelic differences present in a diploid genome form “bubbles” in the assembly graph. Typically assemblers will find and remove these structures to output a linear contig, with one of the two possible allelles chosen to represent the locus in the contig. Most assemblers will catalog these structures for later consideration as candidate variants[1]. However, for a diploid genome this will only find heterozygous sites. The Cortex program [2012] uses multi-colored de Bruijn graphs to directly compare the sequences of two or more genomes - each color in the graph represents a single individual. Walks following a single color in the graph provide partial assemblies of a single individual. By analyzing the pattern of colors through bubbles in the graph, variants can be found and attributed to particular individuals.

The method I discuss below is conceptually similar to that of Cortex with the major difference being that we use the FM-index as the underlying representation of the assembly graph, instead of explicitly using a hash-table based de Bruijn graph with a fixed k-mer size. As described in Chapter 2, the FM-index can represent both the de Bruijn graph (for all k up to the read length) and the string graph. We use this property to develop both de Bruijn graph and string graph-based variant detection algorithms. Additionally, we can use the fact that the FM-index stores the complete sequence of each read to extract all reads harboring a potential variant and use them as input into a Bayesian model to distinguish between true variants and sequencing errors.

The remainder of this chapter will describe the algorithms I have developed. In the following chapter I demonstrate the versatility of these algorithms by finding polymorphisms present in a human population, finding *de novo* mutations acquired by a child with respect to its parents and to discover mutations occurring in a tumour with respect to an individual’s inherited genome.

4.1.1 Collaboration Note

The methods described in this chapter were developed in collaboration with Cornelis Albers. The variant detection, haplotype assembly, haplotype alignment

[1]ABySS and SGA write the sequences of the bubbles to a file, ALLPATHS-LG uses a marked-up FASTA file to describe the ambiguity in the assembly.

and read extraction algorithms are by the author. The probabilistic realignment model described in section 4.3 was developed by Cornelis Albers. Its description is included in this text to complete the description of the variant calling model.

4.2 Algorithms

We use the FM-index to represent the assembly graphs formed from the sets of sequencing reads. We will typically compare two sets of sequences against each other. We will call one set of sequences the *control* set and one set of sequences the *variant* set. We will call the underlying genomes G_c and G_v, respectively. The *variant* sequences will always be a set of sequence reads drawn from G_v. We will denote this read set as $\mathcal{R}_v$. The control sequences can either be a set of reads or the chromosomes of a reference genome. Below, we will describe the algorithms in general terms to cover both cases, with minor modifications that will be stated. We will refer to the set of control sequences in general as $\mathcal{C}$.

Our goal is to determine the loci in G_v that differ with respect to G_c. When a reference genome G_r is available, we will use it to provide a common coordinate system to describe the differences between G_v and G_c. The differences between one of the genomes and G_r will be described in terms of changes to G_r with tuples of the form:

<`reference-position, reference-sequence, variant-sequence`>[1]

These tuples encode the information required to locally change G_r into G_v at the variant site. Let $\mathcal{A}_{vr}$ be the set of tuples describing differences between G_v and G_r and $\mathcal{A}_{cr}$ be the corresponding set of tuples describing differences between G_c and G_r. The set of positions we are interested in finding is $\mathcal{B} = \mathcal{A}_{vr} - \mathcal{A}_{cr}$. When the control genome is the reference genome, this is just $\mathcal{A}_{vr}$.

We begin by building an FM-index for each of $\mathcal{R}_v$ and $\mathcal{C}$ using the methods presented in Chapter 3. We load the pair of FM-indices into memory, then our algorithm has four stages. First, we find a set of substrings that are present in $\mathcal{R}_v$ but not $\mathcal{C}$. These substrings may cover locations in G_v that are different with respect to G_c. We then extend these substrings into candidate haplotypes using

[1]This is information is typically encoded in a Variant Call Format (VCF) file

the joint assembly graph of $\mathcal{R}_v$ and $\mathcal{C}$, as represented by the pair of FM-indices. We then align the candidate haplotypes to G_r. Finally, we use a probabilistic model to assess whether the assembled haplotypes represent true differences between G_v and G_c or sequencing errors. The tuples for variants that are called by our probabilistic model are output to a VCF file. Each stage is discussed in detail below.

4.2.1 Motivating Example

Consider the simple case where G_v and G_c are random genomes that differ at a single position. For some k large enough to avoid spurious matches between unrelated sequence, let $S_v = K_1xK_2$ and $S_c = K_1yK_2$ be the $2k+1$ substrings of G_v and G_c surrounding this single difference. $S_v[1,k] = S_c[1,k] = K_1$ and $S_v[k+2, 2k+1] = S_c[k+2, 2k+1] = K_2$ are the k-mers that occur immediately before and after the single difference. These k-mers are shared between G_v and G_c. The k-mers covering x and y are unique to G_v and G_c. There are k such k-mers, which are the substrings $S_v[1+i, 1+i+k]$ for all $i \in \{1..k\}$ (respectively, S_c). Under our assumption that k is sufficiently large, then $S_v[1+i, 1+i+k]$ k-mers are unique to G_v. It is this set of k-mers that we wish to find as the set of candidate variant k-mers. This situation is depicted in figure 4.1.

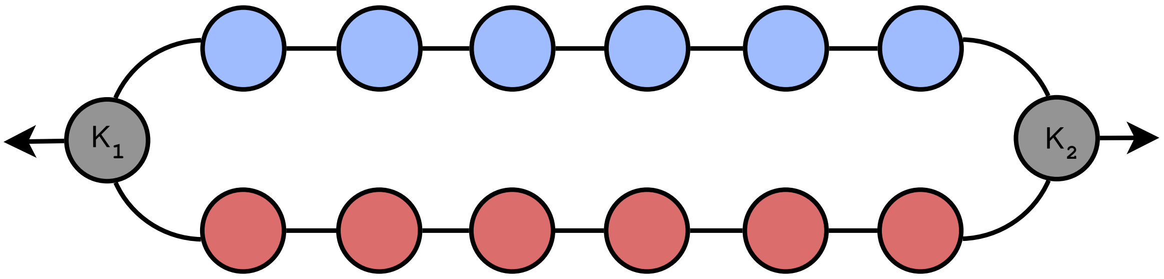

Figure 4.1: A bubble in a de Bruijn graph built from G_v and G_c. The grey k-mers (labelled K_1 and K_2) are shared between G_v and G_c and are the entry/exit points of the bubble. The red and blue vertices represented k-mers unique to G_v and G_c, respectively.

Once we have found the candidate variant k-mers, we assemble them into haplotypes. In the context of the de Bruijn graph[1] shown in figure 4.1, we would start the haplotype generation process from one of the six red k-mers. The haplotype generation would perform a breadth-first search by starting from a red node and continue until both grey junction nodes have been found. The set of nodes found during this search would define a path through the red half of the bubble. We can calculate the assembly string corresponding to this path and output it as a candidate haplotype. We can perform an additional search between the two grey junction nodes using just the control sequences. This will search the blue half of the bubble and generate candidate haplotypes in the control sequences. It is worth noting that these haplotypes may also be present in the variant sequences.

After haplotypes have been generated, we align them to the reference genome then extract all the raw sequence reads from the FM-index that share a k-mer with a candidate haplotype. The candidate haplotypes, their alignments to the reference and the raw sequence reads are input into the Bayesian model which assesses the evidence for each haplotype and makes the final variant calls.

4.2.2 Discovering Candidate Variants

The first stage of the algorithm attempts to find k-mers of G_v that are not present in G_c. If we know the sequence of G_v and G_c this problem is easy. We could decompose the genomes into their k-mer sets $\mathcal{K}_v$ and $\mathcal{K}_c$ and compute the set $\mathcal{D} = \mathcal{K}_v \setminus \mathcal{K}_c$. Of course, we do not know the full sequence of G_v so we must approach the problem from a different direction. In Pevzner's original paper on de Bruijn graph assembly [2001], he observed that the set of k-mers present in a set of sequence reads drawn from G_v approximates the set of k-mers of G_v itself. In Chapter 3 we used this fact to correct substitution sequencing errors. Here, we use it again to solve the problem of finding k-mers unique to G_v. We could explicitly subtract the k-mers of $\mathcal{C}$ from the k-mers of $\mathcal{R}_v$ but this would require the intermediate storage of the full k-mer sets. Instead, we developed an efficient streaming algorithm.

For the moment we will ignore sequencing errors. The algorithm begins by

[1]A string graph based algorithm is given in section 4.2.4

iterating over all reads in $\mathcal{R}_v$ and all k-mers in each read. For each k-mer, Q, we use the FM-index to count the number of occurrences of Q in $\mathcal{R}_v$ and the number of occurrences in $\mathcal{C}$. If Q only appears in $\mathcal{R}_v$, we emit it as a candidate variant. As the same k-mer may appear in multiple reads, we want to avoid emitting duplicate k-mers. To do this, we use an efficient bit-vector marking procedure. At the program's start, we create a bit vector B with one bit per base in $\mathcal{R}_v$, initialized to zero. When we first visit a k-mer Q, we calculate its suffix array interval $[l_Q, u_Q]$ and set the bit $B[l_Q]$ to be one. Subsequent visits to Q will see that $B[l_Q]$ is one and skip it as it has already been visited. This avoids emitting duplicate candidate variants for the same k-mer and also accelerates the search by avoiding redundant k-mer occurrence queries.

An alternative approach to finding k-mers unique to $\mathcal{R}_v$ can avoid traversing over each read in $\mathcal{R}_v$. We can simulate a breadth-first traversal of the implicit suffix tree represented by the FM-index of $\mathcal{R}_v$, stopping once we have reached depth k. Such a traversal operates over suffix array intervals instead of individual k-mers, and hence visits each distinct k-mer only once without the need of the bit vector. When we visit a k-mer, we can check its count in the FM-index of $\mathcal{C}$ to determine whether it should be emitted as a candidate variant. While this algorithm is cleaner than iterating over every read in $\mathcal{R}_v$, it is faster in practice only for small k. The bit-vector based algorithm is very easy to parallelize using multiple threads. Our implementation uses atomic marking (implemented with compare-and-swap instructions) to allow concurrent updates of the bit vector without requiring locks. This highly parallel implementation makes the k-mer discovery portion of the algorithm very fast in practice.

In the presence of sequencing errors, the above algorithms require minor modifications. As discussed in the previous chapter, sequencing errors generate low-frequency k-mers. To help distinguish between unique k-mers arising from errors and unique k-mers arising from true variants, we set a threshold of d (typically 3-5) on the minimum number of occurrences of Q in $\mathcal{R}_v$ to emit the k-mer as a candidate variant. As an additional filter we require that both Q and $\overline{Q}$ are present in $\mathcal{R}_v$ - this requires that k-mer is seen on both sequencing strands, which helps discard systematic errors [Meacham et al., 2011]. Algorithm `generateCandidateVariant` encapsulates the procedure for finding candidate variants.

Algorithm 12 generateCandidateVariants(k, d, $\mathcal{R}_v$, $\mathcal{C}$) - find candidate variant substrings

```
for all $R \in \mathcal{R}_v$ do
  $n \leftarrow |R| - k + 1$
  for $i = 1 \rightarrow n$ do
    $Q \leftarrow R[i, i + k]$
    if not isMarked($Q$) then
      mark($Q$)
      $v_f \leftarrow$ countOccurrences($Q, \mathcal{R}_v$)
      $v_r \leftarrow$ countOccurrences($\overline{Q}, \mathcal{R}_v$)
      $c \leftarrow$ countOccurrences($Q, \mathcal{C}$) + countOccurrences($\overline{Q}, \mathcal{C}$)
      if $c = 0$ and $v_f > 0$ and $v_r > 0$ and $v_f + v_r \geq d$ then
        emit($Q$)
```

Once the candidate variant k-mers have been found, we attempt to assemble them into haplotypes. We have two procedures for doing this, one which uses a de Bruijn graph and one which uses a string graph. We describe both below.

4.2.3 de Bruijn graph haplotype generation

Let Q be a variant k-mer found during the previous portion of the algorithm. If Q represents a true difference with respect to G_c it will lie on one branch of a bubble in the de Bruijn graph formed from the union of $\mathcal{R}_v$ and $\mathcal{C}$. For each vertex (k-mer) of this de Bruijn graph, we can indicate whether it is a k-mer from $\mathcal{R}_v$, $\mathcal{C}$ or both (see figure 4.1). By definition, Q is a k-mer present only in $\mathcal{R}_v$. The algorithm to assemble Q into a candidate haplotype proceeds by performing a breadth-first search starting from Q. The search proceeds until we find k-mers that are present in both $\mathcal{R}_v$ and $\mathcal{C}$. In the context of figure 4.1 we would start the search on one of the red k-mers, and perform the breadth first search outwards in both directions until one of the grey shared k-mers is reached. These *join* k-mers are the entry/exit points of the bubble. As sequencing errors generate new k-mers and paths in the graph, when searching the graph we ignore k-mers that have not been seen at least m times (typically m is 1 to 3). Pseudocode for this algorithm is presented in `generateDeBruijnHaplotypes`.

The `generateDeBruijnHaplotypes` algorithm begins by initializing an empty de Bruijn graph (line 1) and two empty arrays (lines 2 and 3). The arrays will hold the join vertices, where the two halves of the bubble converge. We perform a breadth-first search for these vertices, starting at Q. Lines 5-7 initialize a direction-specific traversal queue with an element for searching from the prefix (left) side of Q and an element for searching from the suffix (right) side of Q. As de Bruijn graphs can be very complex in repetitive regions, we set a limit on how far we will search before aborting the process (lines 9 and 10). The algorithm then loops over all elements of the queue (lines 12-28). As each node is popped from the queue, its neighbors in the de Bruijn graph are found (lines 15-27) and added as vertices if they meet the minimum coverage parameter of m (lines 23,26). If the vertex is found in both $\mathcal{R}_v$ and $\mathcal{C}$, then the vertex is added to the LEFT or RIGHT join array, depending on the direction of traversal (lines 22-24). If the vertex is only present in $\mathcal{R}_v$, it is enqueued and the main loop starts over. Once the graph exploration phase of the algorithm is complete if we have found left and right join vertices we generate candidate haplotype strings by following the graph through each pair of join vertices (lines 28-33). This uses the function `buildHaplotypes` which takes a pair of vertices in the de Bruijn graph and generates all possible paths between the pair of vertices and returns the corresponding assembly string for each path. The strings generated by this procedure are the candidate haplotypes covering the input k-mer.

After candidate variant haplotypes have been generated, we use a similar procedure to generate candidate haplotypes using the control sequences. We perform a directed search of the de Bruijn graph of the control sequences between each pair of join vertices. The assembly string for each path found during this procedure is added to the set of candidate haplotypes.

Algorithm 13 generateDeBruijnHaplotypes(Q, k, m, $\mathcal{R}_v$, $\mathcal{C}$) - assemble candidate variant into a haplotype

```
init(graph, $Q$)
joins[LEFT] $\leftarrow \emptyset$
joins[RIGHT] $\leftarrow \emptyset$
queue $\leftarrow \emptyset$
append(queue,kmer=$Q$, direction=LEFT)
append(queue,kmer=$Q$, direction=RIGHT)
iterations $\leftarrow 0$
max_iterations $\leftarrow 10000$

while queue not empty and iterations $<$ max_iterations do
    $n \leftarrow$ pop(queue)
    $S \leftarrow n.kmer$
    for all $b \in \{A, C, G, T\}$ do
        if $n$.direction is LEFT then
            $T \leftarrow bS[1, k-1]$
        else
            $T \leftarrow S[2, k]b$
        $v \leftarrow$ countOccurrences($T, \mathcal{R}_v$) + countOccurrences($\overline{T}, \mathcal{R}_v$)
        $c \leftarrow$ countOccurrences($T, \mathcal{C}$) + countOccurrences($\overline{T}, \mathcal{C}$)
        if $v \geq m$ and $c > 0$ then
            addDBGVertex(graph, $T$, BOTH)
            append(joins[$n.direction$], $T$)
        else if $v \geq m$ then
            addDBGVertex(graph, $T$, $n$.direction)
            append(queue,kmer=$T$, direction=$n$.direction)
    iterations $\leftarrow$ iterations +1

haplotypes $\leftarrow \emptyset$
if joins[LEFT] not empty and joins[RIGHT] not empty then
    for all $l \in$ joins[LEFT] do
        for all $r \in$ joins[RIGHT] do
            push(haplotypes, buildHaplotypes(graph, $l$, $r$)
return haplotypes
```

4.2.4 String graph haplotype generation

The second haplotype generation function uses the string graph. The string graph haplotype generation algorithm is a composition of algorithms described previously in chapters 2 and 3. Like in Chapter 3, we require all reads to be error corrected before inserting them in the graph. Likewise, we only allow exact overlaps between reads. Unlike our whole genome assembly algorithm we do not error correct the full read set. When G_v and G_c are closely related it is expected that they will have very few differences. In this case it would be inefficient to error correct every read, as most would not harbor variation. Instead, we correct each read as it is processed by the algorithm. The algorithm is described at high level in `generateStringGraphHaplotypes`.

We begin by initializing an empty graph, and arrays to hold join vertices. We extract all reads containing the input k-mer Q from the FM-index of $\mathcal{R}_v$. This set of reads is error corrected using the k-mer correction method described in Chapter 3. The corrected reads are inserted into the graph, and exact overlaps between the vertices are computed. Here, we simply use a hash of τ-mer sequences to compute candidate overlaps. The main loop of the algorithm finds "tip" reads in the graph - those that only have a neighbor on one side (a prefix neighbor or suffix neighbor). Reads sharing a substring with a tip vertex are extracted from the FM-Index (by `findNewOverlaps`) and corrected. The newly corrected reads are then added into the graph. As each read is inserted into the graph, we determine if it is a join vertex. If the k-mer at the start of read X occurs in both $\mathcal{R}_v$ and $\mathcal{C}$, we say that X is a left-join vertex. If the k-mer at the end of read X occurs in both $\mathcal{R}_v$ and $\mathcal{C}$, we say that X is a right-join vertex. After all new vertices have been added to the graph, we run Myers' transitive reduction algorithm [Myers, 2005] on the graph. We then attempt to find walks from the left-joins to the right-joins that cover the reads containing the candidate variant k-mer Q. If these reads are covered by walks, the walks are returned as the candidate haplotypes. As in `generateDeBruijnHaplotypes` we set a bound of max_iterations on the number of times to extend the graph before aborting.

Finally, we generate haplotypes for the control sequences using the same procedure as section 4.2.3. Here, we do not explicitly have the set of join vertices in the implicit de Bruijn graph. Instead, we use the first and last k-mer of each variant candidate haplotype to seed the directed search through the de Bruijn graph.

Algorithm 14 generateStringGraphHaplotypes(Q, k, τ, $\mathcal{R}_v$, $\mathcal{C}$) - assemble candidate variant into a haplotype

```
1: init(graph)
2:
3: joins[LEFT] ← $\emptyset$
4: joins[RIGHT] ← $\emptyset$
5: iterations ← 0
6: max_iterations ← 1000
7:
8: $I \leftarrow$ extractReads($Q, \mathcal{R}_v$)
9: $I_c \leftarrow$ correctReads($I, \mathcal{R}_v$)
10: for all $r \in I_C$ do
11:     addStringVertex(graph, $r$)
12: while iterations < max_iterations do
13:     $T \leftarrow$ findGraphTips(graph)
14:     if $T$ is $\emptyset$ then
15:         return $\emptyset$
16:     for all $t \in T$ do
17:         $O \leftarrow$ findNewOverlaps(graph, $t$, $\tau, \mathcal{R}_v$)
18:         $O_c \leftarrow$ correctReads($O, \mathcal{R}_v$)
19:         for all $o \in O_c$ do
20:             addStringVertex(graph, $o$)
21:             if isLeftJoin($o, k, \mathcal{C}$) then
22:                 push(joins[LEFT], $o$)
23:             if isRightJoin($o, k, \mathcal{C}$) then
24:                 push(joins[RIGHT], $o$)
25:     myersTransitiveReduction(graph)
26:     if joins[LEFT] $\neq \emptyset$ and joins[RIGHT] $\neq \emptyset$ then
27:         haplotypes ← findHapWalks(graph, joins[LEFT], joins[RIGHT])
28:         if haplotypes $\neq \emptyset$ then
29:             return haplotypes
30:     iterations ← iterations + 1
```

4.2.5 Haplotype quality control

After we generate candidate haplotypes we perform a quality check. For a haplotype string H and a set of reads, let k_{max} be the largest k such that all k-mers in H are seen at least l times in the reads. In other words, all k_{max}-mers in H are seen at least l times in the FM-index but some $(k_{max}+1)$-mers of H are not found l times. We expect that haplotypes that are truly present in a genome and well-covered by sequence reads will have a large value k_{max}. Conversely, if a haplotype is not present in a genome, k_{max} will be very small as it will require random k-mer matches to find covering k-mers (we would expect k_{max} to be $\approx \log(|G|)$ for a random haplotype not present in a genome). We can use these observations to define a quality check on the haplotypes that we assembled above. For a haplotype H, let v be k_{max} for the haplotype in the variant read set $\mathcal{R}$. Let c be the corresponding value for k_{max} for the control sequences. We filter out haplotypes when $c \geq 31$ or when $v - c < 10$. The first check ($c \geq 31$) indicates that the haplotype is well-supported in the control sequence set. In this case it is unlikely that it represents a true difference between G_v and G_c. The second check requires the support for a haplotype to be significantly stronger in the variant sequences than the control sequences. The parameter of l (the number of times each k-mer must be seen) is determined by the control sequences. If we are calling variants between two sets of reads, we use $l = 2$ (every k-mer must be seen twice). If we are calling variants against a reference genome we use $l = 1$.

4.3 Probabilistic realignment

To distinguish between sequencing errors and true variants, we use a probabilistic model to determine how well each candidate haplotype is supported by the raw read sequences. Our FM-index based approach easily allows this, as we are able to efficiently extract the full sequence of each read from the index. Our realignment method begins by extracting reads from the FM-index that may match one of the assembled candidate haplotypes. These reads, along with the candidate haplotypes, are the input into our Bayesian model.

4.3.1 Extracting Haplotype Reads from the FM-Index

Extracting a single indexed read from the FM-index is straightforward. Let $\mathcal{R}_i$ be the read in the indexed sequence collection whose sequence we wish to extract. From the definition of the BWT in section 2.4, we know that the suffix array interval for the empty suffix of $\mathcal{R}_i$ is $I = [i, i]$. Correspondingly, the last base of $\mathcal{R}_i$ is given by $\mathbf{B}_{\mathcal{R}}[i]$. Let this base be denoted by b. We can use the function `updateBackward`(I, b) from section 2.4 to calculate the suffix array interval for the one-base suffix of $\mathcal{R}_i$, consisting of the string $b\$$. The corresponding character in the BWT gives the second-last base of $\mathcal{R}_i$. If we iterate this procedure until we reach the \$ symbol in the BWT string, we will have extracted the complete sequence of $\mathcal{R}_i$, as desired.

The procedure to extract haplotype reads is based on k-mer matches. Let H be a haplotype that we wish to find reads for. Let $K_1, K_2, K_3 ... K_n$ be the sequence of k-mers for a haplotype H. We use the FM-index (of $\mathcal{R}_v$ or $\mathcal{C}$) to find suffix array intervals for each of these k-mers. From these k-mer intervals, we backtrack in the FM-index until we reach the terminating \$ symbols. Once the dollar symbols are reached, we use the lexicographic index (section 2.5.1) to map from the lexicographic order of a read to its numeric index in $\mathcal{R}$. These numeric indices are then used in the procedure described in the previous paragraph to extract the full read sequence.

As some reads will share multiple k-mers with a haplotype, the procedure described above is inefficient. To account for multiple k-mers we cache visited intervals during backtracking. If a previously visited interval is visited during backtracking, we exclude that position from further consideration.

For each candidate haplotype we extract the reads from both $\mathcal{R}_v$ and $\mathcal{C}$ matching the haplotype. The set of haplotypes and their matching raw sequence reads are passed to the probabilistic model.

When performing multi-sample calling, like when calling variants present in a low-coverage population of individuals, we need to associate with each read the sample that it originated from. To do this, we create a single FM-index from all samples. We construct the read set $\mathcal{R}$ such that all the reads for sample i are before all reads for sample j. We can then build a simple interval index associating a range of indices in $\mathcal{R}$ with which sample those reads came from. When extracting read i from the FM-index, we can then return the sample identifier along with the read sequence.

4.3.2 Probabilistic read-haplotype alignment

The purpose of realigning reads to candidate haplotype is to obtain the likelihoods $P(\mathbf{R}_i|\mathbf{H}_j, \theta)$. Here, $\mathbf{R}_i$ is the sequence of read i as generated by the sequencing machine and extracted from the FM-index in the previous section, $\mathbf{H}_j$ is the sequence of candidate haplotype j assembled in section 4.2.3 or 4.2.4, and θ is the vector of model parameters. As we do not currently use quality scores for the read bases the model parameters include an assumption that each read base is Q20. The parameters also include homopolymer sequencing error indel rates as described in [Albers et al., 2011]. These read-haplotype likelihoods are combined with a suitable prior probability distribution[1] over the haplotypes to infer which haplotypes are present in a sample or population of samples. The model that underlies the likelihood $P(\mathbf{R}_i|\mathbf{H}_j, \theta)$ is the Bayesian network described previously [Albers et al., 2011]. Here we use a fast approximate version of this model. The approximation consists of testing only two seed alignments rather than all possible alignments. The two seed alignments are computed using a 8-base hash of the read and haplotype sequence.

4.3.3 Annotating variants in the candidate haplotypes

The strategy for calling sequence variation in a reference-free fashion is to first determine which haplotypes are supported by the data, and only then to annotate the haplotypes with respect to a particular coordinate system or reference sequence. In principle the alignment of haplotypes to a reference sequence is a post-processing step. However, there are several advantages of having haplotype mapping locations available during the inference of the haplotypes. The confidence in a variant call depends on whether the haplotype(s) containing the variant is supported by the data, and whether the haplotype can be confidently

[1]The choice of prior probability distribution depends on whether we are calling variants by comparing two genomes or multiple individuals sequenced at low coverage

placed onto the reference. If one of these two factors is uncertain the variant call quality will be low. Furthermore, it is desirable to have available a number of statistics for each variant call that can be used for filtering. For instance, it is useful to know how many reads cover the variant without any mismatch. To be able to provide this information it is necessary to know all the possible mapping locations of a haplotype to a given reference sequence. To the end-user it may be also be useful to know that a novel haplotype is strongly supported by the data but cannot be confidently placed.

4.3.4 Aligning haplotypes to a reference genome

We align the candidate haplotypes to the reference genome, G_r. Our alignment method uses the FM-index of G_r to find l-mer seed matches between each haplotype and the reference genome G_r. These candidate alignments are refined by dynamic programming. During dynamic programming, we require a semi-global alignment between the haplotype and the reference (we require an end-to-end alignment of the haplotype but a local alignment to the reference). We do not require the alignment of the haplotype to the reference genome to be unique. For each candidate alignment, we calculate the number of edit *events* in the alignment. An edit event is a contiguous stretch of differences in the alignment between the haplotype and the reference (for example a 5bp deletion counts as one event, not 5). We keep all alignments that have fewer than 9 edit events. For a repetitive haplotype this may result in multiple locations with a reasonable alignment score.

Each alignment location may result in a different set of variants. We also compute a mapping quality for each mapping location using the haplotype-reference alignment scores. This mapping quality will be used in the calculation of the variant qualities as described below.

4.3.5 Comparative variant-calling

In comparative variant calling we have reads for both G_v and G_c and we wish to detect variants that are only found in G_v but not G_c. A primary application of comparative variant calling is finding somatically acquired mutations in a cancer from a sequenced tumour-normal pair. We describe our comparative variant calling model in these terms. Since a tumour sample is not clonal and many contain entire chromosome duplications or loss, one can not assume a diploid model. We therefore assume that the number of haplotypes present in the tumour sample can be greater than two. For simplicity we made the same assumption for the normal sample.

To deal with a possibly large number of haplotypes, we apply a model selection approach to infer which haplotypes are supported by the reads. In this model selection approach, haplotypes are iteratively added until the improvement to the total score is below the minimum threshold required for adding a new haplotype. After the model selection algorithm has converged, the haplotype frequencies are estimated using the Expectation-Maximization algorithm [Dempster et al., 1977].

The scores for the haplotypes used in the model selection are defined as follows. The increase to the total score by adding a candidate haplotype j to the model is given by

$$\Delta S_j = \sum_i \left(\log P(\mathbf{R}_i|\mathbf{H}_j, \theta) - s_i\right), \tag{4.1}$$

where

$$s_i = \underset{k \in \text{selected haplotypes}}{\arg\max} \log P(\mathbf{R}_i|\mathbf{H}_k, \theta). \tag{4.2}$$

For the first iteration, when no haplotypes have been selected yet, s_i is set to a default minimum score. This minimum score is approximately $\log 10^{-6}$ (Q60), so that in practice a read-haplotype alignment with more than indel (penalty of Q40 outside homopolymer runs) or four mismatches (Q20 per mismatch) will not be above this minimum threshold. This minimum score prevents reads that do not have a reasonable alignment to any of the candidate haplotypes from favoring one haplotype over the other because of irrelevant differences in the read-haplotype likelihood.

To estimate the probability that a candidate variant is a somatic variant, a joint set of candidate haplotypes is created from the candidate haplotypes detected in the normal sample and the candidate haplotypes detected in the tumour sample. Conditional on the joint set of candidate haplotypes, inference in the normal and the tumour sample can be performed independently.

The quality scores for a somatic variant v are next computed as follows:

$$P(v \text{ is somatic}|\mathbf{R}_{\text{normal}}, \mathbf{R}_{\text{tumour}}) = P(v \text{ is present}|\mathbf{R}_{\text{tumour}})P(v \text{ is absent}|\mathbf{R}_{\text{normal}}), \tag{4.3}$$

where $P(v \text{ is present}|\mathbf{R}_{\text{tumour}})$ is the probability that a haplotype containing the variant v is present in the tumour, and $P(v \text{ is absent}|\mathbf{R}_{\text{normal}})$ is the probability that there is no haplotype containing the variant v in the normal sample. The quality score for a variant being present in a sample is calculated as follows:

$$\begin{aligned} P(v \text{ is not present}|\mathbf{R}_{\text{sample}}) &\approx \\ &\prod_{j \in \text{selected}, \mathbf{H}_j \text{contains } v} \Big(1 - P(\mathbf{H}_j \text{ is present}|\mathbf{R}_{\text{sample}})P(\mathbf{H}_j \text{ maps to reference location of } v)\Big) \\ &\approx \prod_j \Big(1 - \big(1 - \exp(-\Delta S_j)\big)P(\mathbf{H}_j \text{maps to reference location of } v)\Big) \end{aligned} \tag{4.4}$$

Thus, the variant quality takes into account both the uncertainty in the presence of the haplotype containing the variant, as well as the uncertainty that each of those haplotypes maps to the location of the variant.

4.3.6 Population calling

The algorithm for population calling is similar to the comparative variant-calling algorithm. The main difference is the calculation of the increase in the score from selecting a haplotype. Instead of Eq. 4.1 we use a multisample EM algorithm to estimate the increase in the likelihood achieved by adding a haplotype j. The log-likelihood for the model consisting of the candidate haplotypes selected in iterations $1, \ldots, k-1$ and candidate haplotype j in iteration k is defined as:

$$\exp L_j^k = \max_{\mathbf{f}_{k-1,j}} \prod_i \sum_{h_i^1} \sum_{h_i^2} P(h_i^1|\mathbf{f}_{k-1,j})P(h_i^2|\mathbf{f}_{k-1,j}) \prod_{l \in \text{reads}} \Big(\frac{1}{2}P(\mathbf{R}_i^l|\mathbf{H}_{h_i^1}, \theta) + \frac{1}{2}P(\mathbf{R}_i^l|\mathbf{H}_{h_i^2}, \theta)\Big) \tag{4.5}$$

Here h_i^1 and h_i^2 are indicator variables for the two haplotypes present in sample i; we explicitly assume a diploid model. $P(\mathbf{R}_i^l|\mathbf{H}_{h_i^1}, \theta)$ is the read-haplotype likelihood computed by the probabilistic realignment algorithm for read l from individual i. $\mathbf{f}_{k-1,j}$ is the vector of haplotype frequencies that is estimated using the EM algorithm. The frequencies $\mathbf{f}_{k-1,j}$ are optimized subject to the constraint that only the haplotypes selected in iterations $1, \ldots, k-1$ and the candidate haplotype j can have non-negative values; other candidate haplotypes (not yet added to the model) are set to zero.

We then define the score as:

$$\Delta S_j = L_j^k - L^{k-1}, \tag{4.6}$$

with L^{k-1} the log-likelihood of Eq. 4.5 for the candidate haplotype added at iteration $k-1$. Finally, in iteration k we add the candidate haplotype with the largest ΔS_j to the model. Candidate haplotype are added to the model until there is no candidate haplotype with a score ΔS_j above the threshold.

4.4 Discussion

In this chapter I described a framework for performing assembly based variant calling with a probabilistic model. There are a number of improvements to this model that could be made in the future. In section 4.2.2 we assume that a variant k-mer does not appear in the control sequence set. In the case of high-depth sequence data, there may be sequencing errors that generate k-mers in the control sequences that match the variant k-mers by chance. These erroneous k-mers may mask the presence of variant k-mers and cause our model to miss variants. In practice this is not a significant problem because there is redundancy in the k-mer detection step, as up to k k-mers may contain the variant sequence - we will detect the variant k-mer if any of these is unique to the variant sequence set. When k is greater than half the read length, the same sequencing error would need to occur in multiple reads to mask all of these k-mers. Despite this redundancy in detection, we are likely to lose some variant calls due to errors, therefore this is a possible point of improvement.

In our probabilistic model, we do not use the per-base quality scores output by the sequencing instrument. An obvious point of improvement is to incorporate these into our model. Quality scores are typically encoded using a single ASCII character, which requires one byte per base. If we naively recorded the quality scores for each base, this amount of memory would be far larger than the size of the FM-index to store the reads. In the future we intend to investigate other means of storing and accessing the quality values, including compressed representations (for example, Huffman coding) or downsampling the quality scores to a smaller range (for example using 2 bits per score by quantizing the scores to 4 levels).

Two recently published programs also take an assembly-based approach to variant calling. Cortex [Iqbal et al., 2012] builds a colored de Bruijn graph from the sequence reads from multiple individuals. It then searches for diverging paths in the graph, which are assembled into haplotypes. The haplotypes are mapped to the reference genome in a post-processing step. While my fundamental approach - finding divergent paths through an assembly graph built from multiple individuals - is similar to Cortex there are a number of important differences. The FM-index represents all de Bruijn graphs for k up to the read length. This allows flexibility in parameter choice as the graph does not need to be reconstructed for every k. Cortex represents the graph as a fixed hash table of k-mers and therefore needs to construct a new graph for every k that is used. The FM-index also allows string graph-based haplotype generation, as demonstrated in section 4.2.4[1]. Finally, the FM-index provides access to the full read sequences, allowing the haplotypes to be assessed in our probabilistic model after assembly (section 4.3).

Fermi [Li, 2012] uses modified versions of the algorithms in Chapter 2 and 3 to assemble reads into contigs using a string graph. After assembly the contigs are aligned to a reference genome and variants are parsed from the alignments. Fermi performs full assembly, in contrast to Cortex and the algorithms described in this chapter which only assemble the haplotypes that are expected to contain variation. The author of Fermi demonstrates impressive performance for human genome variation detection, with SNP calling sensitivity approaching that of mapping-based methods. However at this time Fermi is limited to single samples and does not support comparing multiple individuals.

[1]In the following chapter the difference in performance between the de Bruijn graph and string graph based approaches will be explored