Local Assembly in HaplotypeCaller and Mutect

David Benjamin*†
Broad Institute, 75 Ames Street, Cambridge, MA 02142
(Dated: August 11, 2017)

The GATK tools `HaplotypeCaller` and `Mutect` assemble reads aligned within a window of several hundred base pairs into a de Bruijn graph of local variation. Local haplotypes correspond to paths in this de Bruijn graph, and the downstream likelihood calculations in both tools involve aligning reads to these haplotypes. Here we describe how this de Bruijn graph is generated.

I. CORRECTING READS

Read error correction is turned off by default in `HaplotypeCaller` and `Mutect` and we are not familiar with it, nor have we validated it. Nonetheless, the code exists and can be turned on. The idea is as follows: large kmers are good because they contain more phasing information and therefore yield a simpler de Bruijn graph. For example, two SNVs 20 bases apart with sequenced with error-free reads yield a de Bruijn graph with $2 \times 2 = 4$ paths if $k = 10$ but only 2 paths if $k = 40$, because the latter spans the phased SNVs. For example, consider a reference sequence TGAAACGTATTTGGG and an alt sequence with two phased SNVs, TGAAA(C→T)GTA(T→C)TTGGG. If $k = 3$ no kmer spans both SNVs and so the assembly graph containing two bubbles, one for each SNV, as in Figure 1. If $k = 5$, a kmer spans both SNVs and thus only two paths exist in the assembly graph, as in Figure 2.

One drawback is that larger kmers are more liable to be lost due to sequencing error simply because they contain more bases[1]. The idea of read error correction is to rescue kmers from the occasional error.

First, the `ReadErrorCorrector` kmerizes every read and counts the occurrences of each kmer. Then it builds a map from kmers to their corrected versions where kmers that appear often[2] map to themselves and kmers that appear only once[3] map to their nearest neighbor in Hamming distance[4] within a maximum of two mismatches[5]. Kmers that appear between 1 and 20 times are not part of the correction map.

Then, for every base in every read, the `ReadErrorCorrector` queries the kmer correction map for each overlapping kmer. For example, to correct the fourth base of read ACGTATTC if k = 3, it looks at the corrections for CGT, GTA, and TAT at their third, second, and first positions, respectively. If and only if the corrections are unanimous, the base is corrected. Note that corrected reads are used only for assembly and the original reads are used downstream.

II. BUILDING THE GRAPH

Next, the `ReadThreadingAssembler` assembles the (corrected) reads over several different kmer sizes specified by the `kmerSize` argument[6]. If the given kmer sizes fail to produce a graph without cycles, or if more than 1/5 of the

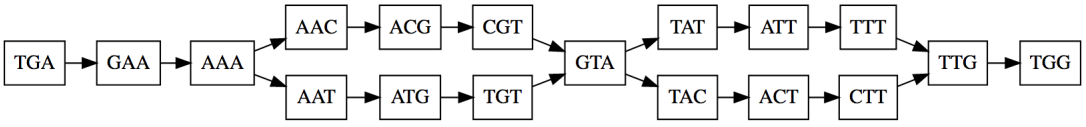

FIG. 1: Two haplotypes yielding assembly graph with two bubbles and four paths due to too-small k.

---

* The author took no part in development of the methods described below – credit belongs to several others on the GATK team.
†Electronic address: `davidben@broadinstitute.org`
[1] Another drawback is that longer kmers yield more dangling heads and tails in the graph (see below), especially near the ends of baited regions in exome sequencing.
[2] By default, 20 times or more. This threshold is set by the `minObservationsForKmerToBeSolid` command line argument.
[3] This is a hardcoded threshold.
[4] For simplicity we consider only substitution errors, eg the distance between ACGGT and AGGTG is 3
[5] This is a hardcoded threshold.
[6] By default, 10 and 25. This size is a compromise and no single value is the best choice for all regions. Large kmers are more likely to be unique and to yield a graph with no cycles, which is especially important in low-complexity regions, but they are more sensitive to

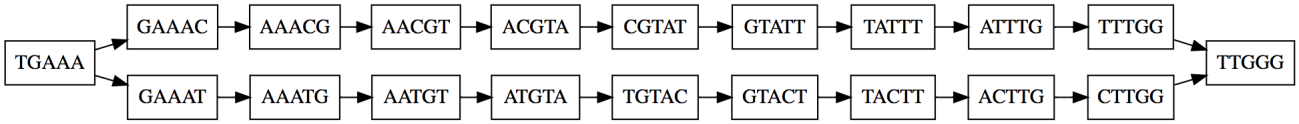

FIG. 2: Two haplotypes yielding assembly graph with a single bubble when k is sufficient to phase variants.

kmers in the graph are not unique in the sequences from which they come[7], the `ReadThreadingAssembler` repeatedly increases k by 10 bases until assembly succeeds or until k has been increased 6 times, that is, by 60 bases[8].

The first step is to create a set of sequences to be kmerized and put into the de Bruijn graph. These sequences are: the reference haplotype, any requested alleles given by the `alleles` argument in `HaplotypeCaller`'s "genotype given alleles" (GGA) mode[9], and maximal subsequences of reads with base quality at least 10 by default[10] and at least one kmer long. That is, a read with 100 bases and a base of quality 7 at position 70 yields sequences of length 69, 30 if $k = 10$ and a single sequence of length 69 if $k = 40$. We do not use mate information; that is, a read and its mate yield completely independent sequences, nor do we later post-process the graph using mate information. This is a potential direction for improvement.

Before building the graph, the `ReadThreadingAssembler` finds the set of all kmers that appear twice in the same sequence i.e. twice in the reference or twice in any read. Then each sequence is kmerized starting from the first kmer that is not in this set. When a new kmer appears, a vertex is added to the graph; otherwise an edge is added (or the multiplicity of an existing edge incremented) from the preceding kmer to the current one. Up to this point, we have produced a de Bruijn graph in textbook fashion, with the exception that we have excluded kmers containing low-quality bases and non-unique kmers at the beginning of reads. Note that this is a directed graph of kmers from aligned reads (i.e. aligned to the forward strand of the reference), not a bidirected graph of unaligned reads that could come from either strand, with the complications of associating a kmer with its reverse complement.

III. CLEANING THE GRAPH

Before deciding on candidate haplotypes, the assembler simplifies the graph with the following heuristics to remove spurious paths and to merge variant paths that diverge from the reference.

- pruning: The assembler finds all maximal non-branching subgraphs and removes those that 1) do not share an edge with the reference path and 2) contain no edges with sufficient multiplicity[11] While the default multiplicity threshold of 2 is quite permissive, it *does* cause `Mutect` to lose sensitivity for deletions occurring in a single read[12].

- dangling tails: The assembler only outputs haplotypes that start and end with a reference kmer, so it attempts to rescue paths in the graph that do not. To rescue a "dangling tails" – a path that ends in a non-reference kmer vertex – the assembler first traverses the graph backwards from this vertex to a reference vertex. If during traversal it encounters a vertex with more than one incoming edge it gives up[13] It also gives up if it encounters a vertex with more than one outgoing edge, that is, if the path branches again after diverging from the reference[14]. Then it generates the Smith-Waterman alignment of the branching path versus the reference path after the vertex at which they diverge. If the alignment's CIGAR contains three or fewer elements, that is, if the alignment has at most one indel, the assembly engine attempts to merge the dangling tail back into the reference.

---

errors and low coverage.

[7] This uniqueness condition is waived on the last attempt when k has been increased by 60 bases.

[8] These constants, 10 and 6, are hard-coded, but the attempts to increase k can be disabled completely with the `dontIncreaseKmerSizesForCycles` argument.

[9] This mode is disabled in `Mutect` and incompatible with the reference confidence mode of `HaplotypeCaller`, which is the recommended best practice. We document it here for completeness.

[10] This is controlled by the `minBaseQualityScore` argument.

[11] By default 2. This is controlled by the `minPruning` argument.

[12] While a SNV occurring on a single read would not yield a confident somatic variant call, a long deletion in a non-STR context could easily be supported by a single read be due to the tiny probability of its arising from sequencing error.

[13] as opposed to doing eg depth-first search of all possible paths back to the reference.

[14] It seems like this could be changed to increase sensitivity.

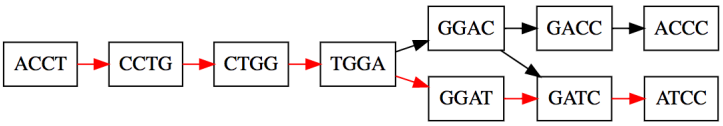

FIG. 3: A dangling tail merged back into the reference path. Reference path edges given by red arrows.

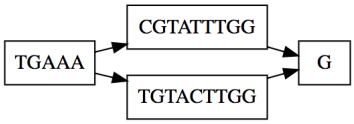

FIG. 4: A zipped sequence graph.

To merge the dangling tail back into the reference path, the assembler finds the beginning of the maximal common suffix of the dangling path and the reference path, that is, the point at which the sequences coverges[15] and adds an edge between the dangling path's vertex and the reference path's vertex at this position. This means that the graph is no longer a valid de Bruijn graph because the dangling vertex kmer and its succeeding reference vertex kmer do not overlap by $k-1$ bases. Nonetheless, this graph yields valid haplotypes when we later "zip" the graph's chains (see below) by accumulating the last base of each kmer.

Figure 3 shows the result of the tail from alt path ACCTGGA(T→C)CC being merged back into the reference path.

- dangling heads: This case and its treatment in the assembly engine is the mirror image of dangling tails.
- non-reference paths: After attempting to merge dangling heads and tails into the reference path, the assembler deletes edges and vertices belonging to dead ends i.e. non-branching subgraphs that start at non-reference source vertices or end at non-reference sink vertices. This is achieved by performing a breadth-first search of vertices moving forward from the reference source and a breadth-first search of vertices moving backwards from the reference sink and keeping only vertices found in both searches.
- zipping chains: A de Bruijn graph of kmers is convenient for performing assembly but is inefficient for summarizing the results of assembly. In this step the assembler converts the de Bruijn graph[16] into a sequence graph by combining all vertices in each maximal non-branching subgraph (i.e. each linear chain) into a single vertex containing the sequence implied by those vertices' kmers[17]. Thus, for example, a de Bruijn graph containing one reference path and one bubble for a single variant, such as the graph of Figure 2 yields a sequence graph with four vertices: the reference subgraphs before and after the bubble and the two sides of the bubble, as in Figure 4.
- merging diamonds: The assembler looks for nodes A and C such that multiple paths $A \to B_i \to C$ exist and absorbs the common prefix of $\{B_i\}$ into A and the common suffix of $\{B_i\}$ into C. For example, if $k = 10$ and there was a single SNV bubble in the de Bruijn graph, the zipped sequence graph has a reference source path

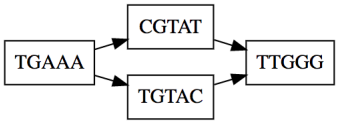

FIG. 5: A merged diamond.

[15] this is *not* where the *paths in the graph* converge (they don't) because kmers in the suffix disagree with the ref at upstream bases.
[16] or rather, if dangling paths have been merged, an *almost* de Bruijn graph.
[17] That is, the concatenation of the last bases of all kmers, except for the first vertex which contributes its entire kmer if and only if it is a source.

(A), a reference sink path (C) and two sides of the bubble (B_1 and B_2) that are each 10 bases long. Since B_1 and B_2 differ only in a single base, their common bases can be absorbed into A and C such that B_1 and B_2 contain only a single base each. Merging tails works the same way without the terminal vertex C and merging common suffixes works the same way without the terminal vertex A. Figure 5 shows the graph of Figure 4 after the common suffix GGTT of the two sides of the bubble are absorbed into the terminal vertex.

- splitting common prefixes and suffixes: If there are multiple nodes S_i with a common predecessor and successor of the form $S_i = A + x_i + B$, where A is a common prefix, B is a common suffix, and x_i is unique to S_i, the assembler splits each S_i such that A and B become independent nodes, with the x_i in between.
- The clean-up steps of zipping chains, merging diamonds, tails, and common suffixes, and splitting common prefixes and suffixes are repeated until no more transformations occur.

When the `debugGraphTransformations` argument is used, the assembly engine emits graphviz .dot files for each of the steps above.

IV. FINDING HAPLOTYPES

At this point there is one cleaned sequence graph for each successful kmer size. For each graph we find the best haplotypes[18] according the following score: the score of a path (haplotype) in a sequence graph is the sum over all branching vertices in the path of the log of the multiplicity of the outgoing edge in the path minus the log of the total multiplicity of all outgoing edges. This formula is implemented as a recursive depth-first search: Starting from the reference source vertex, the haplotype finder adds branching scores and instantiates new haplotype finders for each edge whenever a path diverges and otherwise simply moves to the succeeding vertex without changing the accumulated score. The procedure ends when every haplotype finder reaches a sink vertex, that is, when the exhaustive search of all paths is finished.

This description completely defines the best haplotypes; the actual implementation is a somewhat complicated recursive algorithm with lots of polymorphism.

[18] This is 128 by default and set by the `maxNumHaplotypesInPopulation` argument.