Variant Discovery and Canonicalization via Indexed de Bruijn Graphs

Frank Austin Nothaft, Anthony D. Joseph,
David A. Patterson
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{fnothaft, adj, pattrsn}@berkeley.edu

January 23, 2015

Abstract

Modern variant calling methods frequently rely on local reassembly based approaches to discover variants. Although local reassembly allows for the accurate reconstruction of insertion and deletion variants in the presence of local mapping errors, current reassembly methods provide low throughput. These methods achieve low throughput due to a reliance on an exhaustive graph search to identify haplotypes, coupled with the use of expensive realignment methods for haplotype scoring. In this paper, we enhance the traditional *de Bruijn* graph with the location of k-mers in a reference sequence. First, we demonstrate how an *indexed de Bruijn* graph simplifies to an efficient algorithm for local sequence alignment. We are able to derive canonical edit representations, and achieve $\Omega(n)$ alignment cost for canonical edits. From this alignment formulation, we demonstrate how an *indexed de Bruijn* graph can be used for variant discovery in a local reassembly pipeline with lower cost than a traditional reassembly algorithm. An *indexed de Bruijn* graph has identical storage cost as a traditional *de Bruijn* graph.

1 Introduction

The accuracy of insertion and deletion (INDEL) variant discovery has been improved by the development of variant callers that couple local reassembly with haplotype-based statistical models to recover INDELs that were locally misaligned [1]. Now, several prominent variant callers such as the Genome Analysis Toolkit's (GATK) `HaplotypeCaller` [4], `Scalpel` [10], and `Platypus` [11]. Although haplotype-based methods have enabled more accurate INDEL and single nucleotide polymorphism (SNP) calls [2], this accuracy comes at the cost of end-to-end runtime [13]. Several recent projects have been focused on improving reassembly cost either by limiting the percentage of the genome that is reassembled [3] or by improving the performance of algorithms of the core algorithms used in local reassembly [11].

The performance issues seen in haplotype reassembly approaches derives from the high asymptotic complexity of reassembly algorithms. Although specific implementations may vary slightly, a typical local reassembler performs the following steps:

1. A *de Bruijn* graph is constructed from the reads aligned to a region of the reference genome,

2. All valid paths (*haplotypes*) between the start and end of the graph are enumerated,

3. Each read is realigned to each haplotype, typically using a pair Hidden Markov Model (HMM, see [5]),

4. A statistical model uses the read↔haplotype alignments to choose the haplotype pair that most likely represents the variants hypothesized to exist in the region,

5. The alignments of the reads to the chosen haplotype pair are used to generate statistics that are then used for genotyping.

In this paper, we focus on steps one through three of the local reassembly problem, as there is wide variation in the algorithms used in stages four and five (see §2). Stage one (graph creation) has approximately $\mathcal{O}(rl_r)$ time complexity, and stage two (graph elaboration) has $\mathcal{O}(h \max(l_h))$ time complexity. The asymptotic time cost bound of local reassembly comes from stage three, where cost is $\mathcal{O}(hrl_r \max(l_h))$, where h is the number of haplotypes tested in this region[1], r is the number of reads aligned to this region, l_r is the read length[2], and $\min(l_h)$ is the length of the shortest haplotype that we are evaluating. This complexity comes from realigning r reads to h haplotypes, where realignment has complexity $\mathcal{O}(l_r l_h)$. We ignore the storage complexity of reassembly here, but provide an extended discussion of *de Bruijin* graph complexity in §2.

In this paper, we introduce the *indexed de Bruijn* graph and demonstrate how it can be used to reduce the asymptotic complexity of reassembly. An *indexed de Bruijn* graph is identical to a traditional *de Bruijn* graph, with one modification: when we create the graph, we annotate each k-mer with the index position of that k-mer in the sequence it was observed in. This simple addition enables the use of the *indexed de Bruijn* graph for $\Omega(n)$ local sequence alignment with canonical edit representations for most edits. This structure can be used for both sequence alignment and assembly, and achieves a more efficient approach for variant discovery via local reassembly.

2   Background

Current variant calling pipelines depend heavily on realignment based approaches for accurate genotyping [9]. Although there are several approaches that do not make explicit use of reassembly, all realignment based variant callers use an algorithmic structure similar to the one described in §1. In non-assembly approaches like `FreeBayes` [7], stages one and two are replaced with a single step where the variants observed in the reads aligned to a given haplotyping region are filtered for quality and integrated directly into the reference haplotype in that region. In both approaches, local alignment errors (errors in alignment *within* this region) are corrected by using a statistical model to identify the most likely location that the read could have come from, given the other reads seen in this area.

Although the model used for choosing the best haplotype pair to finalize realignments to varies between methods (e.g., the GATK's `IndelRealigner` uses a simple log-odds model [4], while methods like `FreeBayes` [7] and `Platypus` [11] make use of richer Bayesian models), these methods require an all-pairs alignment of reads to candidate haplotypes. This leads to the runtime complexity bound of $\mathcal{O}(hrl_r \min(l_h))$ given in §1, as we must realign r reads to h haplotypes, where the cost of realigning one read to one haplotype is $\mathcal{O}(l_r \max(l_h))$, where l_r is the read length (assumed to be constant for Illumina sequencing data) and $\max(l_h)$ is the length of the longest haplotype. Typically, the data structures used for realignment ($\mathcal{O}(l_r \max(l_h))$ storage cost) can be reused. These

---

[1]The number of haplotypes tested may be lower than the number of haplotypes reassembled. Several tools (see [4, 7]) allow users to limit the number of haplotypes evaluated to improve performance.

[2]For simplicity, we assume constant read length. This is a reasonable assumption as many of the variant callers discussed target Illumina reads that have constant length.

methods typically retain *only* the best local realignment per read per haplotype, thus bounding storage cost at $\mathcal{O}(hr)$.

For non-reassembly based approaches, the cost of generating candidate haplotypes is $\mathcal{O}(r)$, as each read must be scanned for variants, using the pre-existing alignment. These variants are typically extracted from the CIGAR string, but may need to be normalized [9]. *de Bruijn* graph based reassembly methods have similar $\mathcal{O}(r)$ time complexity for building the *de Bruijn* graph as each read must be sequentially broken into k-mers, but these methods have a different storage cost. Specifically, storage cost for a *de Bruijn* graph is similar to $\mathcal{O}(k(l_{\text{ref}} + l_{\text{variants}} + l_{\text{errors}}))$, where l_{ref} is the length of the reference haplotype in this region, l_{variants} is the length of true variant sequence in this region, l_{errors} is the length of erroneous sequence in this region, and k is the k-mer size. In practice, we can approximate both errors and variants as being random, which gives $\mathcal{O}(kl_{\text{ref}})$ storage complexity. From this graph, we must enumerate the haplotypes present in the graph. Starting from the first k-mer in the reference sequence for this region, we perform a depth-first search to identify all paths to the last k-mer in the reference sequence. Assuming that the graph is acyclic (a common restriction for local assembly, see §5.1), we can bound the best case cost of this search at $\Omega(h \min l_h)$.

The number of haplotypes evaluated, h, is an important contributor to the algorithmic complexity of reassembly pipelines, as it sets the storage and time complexity of the realignment scoring phase, the time complexity of the haplotype enumeration phase, and is related to the storage complexity of the *de Bruijn* graph. The best study of the complexity of assembly techniques was done by Kingsford et al. [8], but is focused on *de novo* assembly and pays special attention to resolving repeat structure. In the local realignment case, the number of haplotypes identified is determined by the number of putative variants seen. We can naïvely model this cost with (1), where f_v is the frequency with which variants occur, ϵ is the rate at which bases are sequenced erroneously, and c is the coverage (read depth) of the region.

$$h \sim f_v l_{\text{ref}} + \epsilon l_{\text{ref}} c \tag{1}$$

This model is naïve, as the coverage depth and rate of variation varies across sequenced datasets, especially for targeted sequencing runs [6]. Additionally, while the ϵ term models the total number of sequence errors, this is not completely correlated with the number of *unique* sequencing errors, as sequencing errors are correlated with sequence context [4]. Many current tools allow users to limit the total number of evaluated haplotypes, or apply strategies to minimize the number of haplotypes considered, such as filtering observed variants that are likely to be sequencing errors [7], restricting realignment to INDELs (`IndelRealigner`, [4]), or by trimming paths from the assembly graph. Additionally, in an *de Bruijn* graph, errors in the first k or last k bases of a read will manifest as spurs (see §5.2) and will not contribute paths through the graph. We provide (1) solely as a motivating approximation, and hope to study these characteristics in more detail in future work.

3 Formulation

To construct an *indexed de Bruijn* graph, we start with the traditional formulation of a *de Bruijn* graph for sequence assembly:

Definition 1 (de Bruijn Graph). *A de Bruijn graph describes the observed transitions between adjacent k-mers in a sequence. Each k-mer s represents a k-length string, with a $k-1$ length prefix given by prefix(s) and a length 1 suffix given by suffix(s). We place a directed edge ($\rightarrow$) from k-mer s_1 to k-mer s_2 if prefix$(s_1)^{\{1,k-2\}}$ + suffix(s_1) = prefix(s_2).*

3

Now, suppose we have n sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Let us assert that for each k-mer $s \in \mathcal{S}_i$, then the output of function $\text{index}_i(s)$ is defined. This function provides us with the integer position of s in sequence $\mathcal{S}_i$. Further, given two k-mers $s_1, s_2 \in \mathcal{S}_i$, we can define a distance function $\text{distance}_i(s_1, s_2) = |\text{index}_i(s_1) - \text{index}_i(s_2)|$. To create an *indexed de Bruijn* graph, we simply annotate each k-mer s with the $\text{index}_i(s)$ value for all $\mathcal{S}_i, i \in \{1, \ldots, n\}$ where $s \in \mathcal{S}_i$. This index value is trivial to log when creating the original *de Bruijn* graph from the provided sequences.

Let us require that all sequences $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are not repetitive, which implies that the resulting *de Bruijn graph* is acyclic. If we select any two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that share at least two k-mers s_1 and s_2 with common ordering ($s_1 \to \cdots \to s_2$ in both $\mathcal{S}_i$ and $\mathcal{S}_j$), the *indexed de Bruijn* graph G provides several guarantees:

1. If two sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ share at least two k-mers s_1 and s_2, we can provably find the maximum edit distance d of the subsequences in $\mathcal{S}_i$ and $\mathcal{S}_j$, and bound the cost of finding this edit distance at $\mathcal{O}(nd)$,[3]

2. For many of the above subsequence pairs, we can bound the cost at $\mathcal{O}(n)$, *and* provide canonical representations for the necessary edits,

3. $\mathcal{O}(n^2)$ complexity is restricted to aligning the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that exist *before* s_1 or *after* s_2.

Let us focus on cases 1 and 2, where we are looking at the subsequences of $\mathcal{S}_i$ and $\mathcal{S}_j$ that are between s_1 and s_2. A trivial case arises when both $\mathcal{S}_i$ and $\mathcal{S}_j$ contain an identical path between s_1 and s_2 (i.e., $s_1 \to s_n \to \cdots \to s_{n+m} \to s_2$ and $s_{n+k} \in \mathcal{S}_i \wedge s_{n+k} \in \mathcal{S}_j \forall k \in \{0, \ldots, m\}$). Here, the subsequences are clearly identical. This determination can be made trivially by walking from vertex s_1 to vertex s_2 with $\mathcal{O}(m)$ cost.

However, three distinct cases can arise whenever $\mathcal{S}_i$ and $\mathcal{S}_j$ diverge between s_1 and s_2. For simplicity, let us assume that both paths are independent (see Definition 2). These three cases correspond to there being either a canonical substitution edit, a canonical INDEL edit, or a non-canonical (but known distance) edit between $\mathcal{S}_i$ and $\mathcal{S}_j$.

Definition 2 (Path Independence). *Given a non-repetitive* de Bruijn *graph G constructed from $\mathcal{S}_i$ and $\mathcal{S}_j$, we say that G contains independent paths between s_1 and s_2 if we can construct two subsets $\mathcal{S}'_i \subset \mathcal{S}_i, \mathcal{S}'_j \subset \mathcal{S}_j$ of k-mers where $s_{i+n} \in \mathcal{S}'_i \forall n \in \{0, \ldots, m_i\}, s_{i+n-1} \to s_{i+n} \forall n \in \{1, \ldots, m_i\}, s_{j+n} \in \mathcal{S}'_j \forall n \in \{0, \ldots, m_j\}, s_{j+n-1} \to s_{j+n} \forall n \in \{1, \ldots, m_j\}$, and $s_1 \to s_i, s_j; s_{i+m_i}, s_{j+m_j} \to s_2$ and $\mathcal{S}'_i \bigcap \mathcal{S}'_j = \emptyset$, where $m_i = distance_{\mathcal{S}_i}(s_1, s_2)$, and $m_j = distance_{\mathcal{S}_j}(s_1, s_2)$. This implies that the sequences $\mathcal{S}_i$ and $\mathcal{S}_j$ are different between s_1, s_2,*

We have a canonical substitution edit if $m_i = m_j = k$, where k is the k-mer size. Here, we can prove that the edit between $\mathcal{S}_i$ and $\mathcal{S}_j$ between s_1, s_2 is a single base substitution k letters *after* $\text{index}(s_1)$:

Proof regarding Canonical Substitution. Suppose we have two non-repetitive sequences, $\mathcal{S}'_i$ and $\mathcal{S}'_j$, each of length $2k + 1$. Let us construct a *de Bruijn* graph G, with k-mer length k. If each sequence begins with k-mer s_1 and ends with k-mer s_2, then that implies that the first and last k letters of $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are identical. If both subsequences had the same character at position k, this would imply that both sequences were identical and therefore the two paths between s_1, s_2 would not be independent (Definition 2). If the two letters are different *and* the subsequences are non-repetitive, each character is responsible for k previously unseen k-mers. This is the only possible explanation for the two independent k length paths between s_1 and s_2. $\square$

---

[3]Here, $n = \max(distance_{\mathcal{S}_i}(s_1, s_2), distance_{\mathcal{S}_j}(s_1, s_2))$.

To visualize the graph corresponding to a substitution, take the two example sequences `CCACTGT` and `CCAATGT`. These two sequences differ by a `C` $\leftrightarrow$ `A` edit at position three. With k-mer length $k = 3$, this corresponds to the graph in Figure 1.
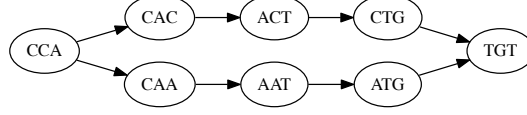

Figure 1: Subgraph Corresponding To a Single Nucleotide Edit

If $m_i = k - 1, m_j \geq k$ or vice versa, we have a canonical INDEL edit (for convenience, we assume that $\mathcal{S}'_i$ contains the $k - 1$ length path). Here, we can prove that there is a $m_j - m_i$ length insertion[4] in $\mathcal{S}'_j$ relative to $\mathcal{S}'_i$, $k - 1$ letters *after* index(s_1):

Lemma 1 (Distance between k length subsequences)**.** *Indexed de Bruijn graphs naturally provide a distance metric for k length substrings. Let us construct an* indexed de Bruijn *graph G with k-mers of length k from a non-repetitive sequence $\mathcal{S}$. For any two k-mers $s_a, s_b \in \mathcal{S}, s_a \neq s_b$, the distance$_\mathcal{S}(s_a, s_b)$ metric is equal to $l_p + 1$, where l_p is the length of the path (in k-mers) between s_a and s_b. Thus, k-mers with overlap of $k - 1$ have an edge directly between each other ($l_p = 0$) and a distance metric of 1. Conversely, two k-mers that are adjacent but not overlapping in $\mathcal{S}$ have a distance metric of k, which implies $l_p = k - 1$.*

Proof regarding Canonical INDELs. We are given a graph G which is constructed from two non-repetitive sequences $\mathcal{S}'_i$ and $\mathcal{S}'_j$, where the only two k-mers in both $\mathcal{S}'_i$ and $\mathcal{S}'_j$ are s_1 and s_2 and both sequences provide independent paths between s_1 and s_2. By Lemma 1, if the path from $s_1 \rightarrow \cdots \rightarrow s_2 \in \mathcal{S}'_i$ has length $k - 1$, then $\mathcal{S}'_i$ is a string of length $2k$ that is formed by concatenating s_1, s_2. Now, let us suppose that the path from $s_1 \rightarrow \cdots \rightarrow s_2 \in \mathcal{S}'_j$ has length $k + l - 1$. The first l k-mers after s_1 will introduce a l length subsequence $\mathcal{L} \subset \mathcal{S}'_j, \mathcal{L} \not\subset \mathcal{S}'_i$, and then the remaining $k - 1$ k-mers in the path provide a transition from $\mathcal{L}$ to s_2. Therefore, $\mathcal{S}'_j$ has length of $2k + l$, and is constructed by concatenating $s_1, \mathcal{L}, s_2$. This provides a canonical placement for the inserted sequence $\mathcal{L}$ in $\mathcal{S}'_j$ between s_1 and s_2. $\qquad\square$

To visualize the graph corresponding to a canonical INDEL, take the two example sequences `CACTGT` and `CACCATGT`. Here, we have a `CA` insertion after position two. With k-mer length $k = 3$, this corresponds to the graph in Figure 2.
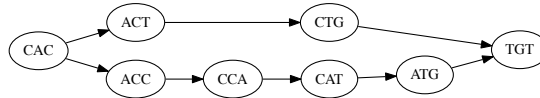

Figure 2: Subgraph Corresponding To a Canonical INDEL Edit

Where we have a canonical allele, the cost of computing the edit is set by the need to walk the graph linearly from s_1 to s_2, and is therefore $\mathcal{O}(n)$. However, in practice, we will see differences that cannot be described as one of the earlier two canonical approaches. First, let us generalize from the two above proofs: if we have two independent paths between s_1, s_2 in the *de Bruijn* graph

---

[4]This is equivalently a $m_j - m_i$ length deletion in $\mathcal{S}'_i$ relative to $\mathcal{S}'_j$.

G that was constructed from $\mathcal{S}_i, \mathcal{S}_j$, we can describe $\mathcal{S}_i$ as a sequence created by concatenating $s_1, \mathcal{L}_i, s_2$.[5] The canonical edits merely result from special cases:

- In a canonical substitution edit, $l_{\mathcal{L}_i} = l_{\mathcal{L}_j} = 1$.

- In a canonical INDEL edit, $l_{\mathcal{L}_i} = 0, l_{\mathcal{L}_j} \geq 1$.

Conceptually, a non-canonical edit occurs when two edits occur within k positions of each other. In this case, we can trivially fall back on a $O(nm)$ local alignment algorithm (e.g., a pairwise HMM or Smith-Waterman, see [5, 12]), *but* we only need to locally realign $\mathcal{L}_i$ against $\mathcal{L}_j$, which reduces the size of the realignment problem. However, we can further limit this bound by limiting the maximum number of INDEL edits to $d = |l_{\mathcal{L}_i} - l_{\mathcal{L}_j}|$. This allows us to use an alignment algorithm that limits the number of INDEL edits (e.g., Ukkonen's algorithm [14]). By this, we can achieve $O(n(d+1))$ cost.

4 Specialization for Local Reassembly

As stated in §3, we have assumed that we want to find the edits between two or more *known* sequences. However, when performing local reassembly for variant discovery/calling, our goal is to identify all possible variants and to associate probabilities to observations that contain these variants. This hypothesized variants are generated by examining the reads aligned to the reference at/near a given site.

However, we can adopt a "pooled" model that uses the *indexed de Bruijn* graph to discover alternate alleles without performing a search for all haplotypes. Here, we extract a substring $\mathcal{R}$ from a reference assembly, corresponding to the subsection of that reference that we would like to reassemble. Then, we create a pooled "sequence" $\mathcal{P}$, that is generated from the k-mers present in the reads aligned to $\mathcal{R}$. However, since $\mathcal{P}$ is a composite of the pooled reads, we cannot assign indices to k-mers in $\mathcal{P}$. Instead, we will rely wholly on the path length properties demonstrated in §3 and the indices of k-mers in $\mathcal{R}$ to discover and anchor alleles. First, let us classify paths where $\mathcal{R}$ and $\mathcal{P}$ diverge into two types:

- **Spurs:** A spur is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where either s_1 or $s_n \in \mathcal{R}, \mathcal{P}$ and all other k-mers are $\notin \mathcal{R}, \in \mathcal{P}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$. If $s_1 \in \mathcal{R}$, then s_n must not have a successor. Alternatively, if $s_n \in \mathcal{R}$, than s_1 is required to not have a predecessor.

- **Bubbles:** A bubble is a set S of n k-mers $\{s_1, \ldots, s_n\}$ where both s_1 and $s_n \in \mathcal{R}, \mathcal{P}$ and all other k-mers are $\notin \mathcal{R}, \in \mathcal{P}$, and where $s_i \rightarrow s_{i+1} \forall i \in \{1, \ldots, n-1\}$.

Currently, we disregard spurs. Spurs typically result from sequencing errors near the start or end of a read. Additionally, given a spur, we cannot put a constraint on what sort of edit it may be from the reference, which increases the computational complexity of processing the spur. We concede that this may not be the best approach, and have included a longer discussion of the accuracy impact of this decision, as well as options for processing spurs with an *indexed de Bruijn* graph in §5.2.

We can elaborate the graph and identify variants by walking the graph from the first k-mer in $\mathcal{R}$. Although haplotype elaboration algorithms have $\Omega(h \min l_h)$ cost (see §2) where $\min l_h$ is the length of the shortest haplotype and h is the number of haplotypes described by the graph, we can

---

[5]This property holds true for $\mathcal{S}_j$ as well.

limit our graph traversal to have $\mathcal{O}(n)$ runtime cost where $n = V(G)$ by introducing a tail-recursive finite state machine (FSM). Whenever we reach a branch point in the graph, our FSM will push state onto a stack, which allows us—with a few exceptions—to avoid making multiple traversals through a single k-mer in the graph. Our FSM has the following states:

- R, `Reference`: We are on a run of k-mers that are in $\mathcal{R}$.

- A, `Allele`: We are on a run of k-mers that have diverged off of the reference. We have a divergence start point, but have not connected back to the reference yet. This could be either a bubble or a spur.

- C, `ClosedAllele`: We were on a run of k-mers that had diverged off of the reference, but have just reconnected back to the reference and now know the start and end positions (in $\mathcal{R}$) of the bubble, as well as the non-reference sequence and length of the bubble.

We allow the FSM to make the following state transitions, which are depicted in Figure 3:

- R $\rightarrow$ R: We are continuing along a reference run.

- R $\rightarrow$ A: We were at a k-mer $\in \mathcal{R}$, and have seen a branch to a k-mer $\notin \mathcal{R}$.

- A $\rightarrow$ A: We are continuing along a non-reference run.

- A $\rightarrow$ C: We were on a non-reference run, and have just connected back to a reference k-mer.

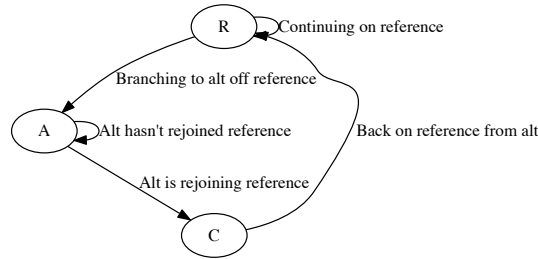- C $\rightarrow$ R: We have just closed out an allele, and are back at a k-mer $\in \mathcal{R}$.

Figure 3: Finite State Machine for Pooled vs. Reference Assembly

We initialize the state machine to R, and start processing with the first k-mer from $\mathcal{R}$. Per k-mer, we evaluate the possible state transitions of each successor k-mer. If the set of successor k-mers contains a single k-mer, we continue to that state. If the successor set contains multiple k-mers, we choose a successor state to transition to, and push the branch context of all other successor states onto our stack. If the successor set is empty, we pop a branch context off of the stack, and switch to that context. We stop once we reach a k-mer whose successor set is empty, *and* our branch context stack is empty.

The implementation of the R and A states largely amount to bookkeeping. In the R state, we must track the current position in $\mathcal{R}$, and in A, we must record where we branched off of $\mathcal{R}$, and $\mathcal{L}'$, the bases we have seen since we branched[6] off of $\mathcal{R}$. If we are in the A state and walk to a

---

[6]Although we are processing k-mers, we only need to reconstruct the sequence that the k-mers are describing by taking the first base from each successive k-mer.

k-mer $\in \mathcal{R}$, we then transition into the C state. Using the positions of our branch point and the current k-mer in $\mathcal{R}$, we are able to calculate the length of the reference path via Lemma 1, while the length of the non-reference path is given by $l_{\mathcal{L}'}$. The edited sequence $\mathcal{L}$ present between the branch point and the current k-mer is given by trimming the first $k-1$ bases from $\mathcal{L}'$. If we have a non-canonical edit, or a deletion in $\mathcal{P}$ relative to $\mathcal{R}$, we can look these bases up by indexing into $\mathcal{R}$.

To improve computational efficiency, we should emit statistics for genotyping *while* traversing the graph. By doing this, we can eliminate the need for realignment in haplotype reassembly. Since we have constructed a graph where every k-mer is anchored to a reference position or to a position in a variant, or is in a spur, if we log several statistics when processing reads from $\mathcal{P}$, we can directly emit the probability observations that support a reference base or a variant allele. Although the relevant statistics vary between different variant calling/genotyping algorithms, common statistics include base and mapping quality, as well as strand of the observation. Additionally, genotyping algorithms that incorporate phasing may want to identify observations that came from the same allele. These statistics can be logged when chopping the reads in $\mathcal{P}$ into k-mers at a low cost.

As noted earlier in this section, we normally do *not* need to retrace through any k-mers in the graph when processing the graph. However, we may need to retrace k-mers if we have overlapping variants. For example, take the reference string ACTGCCGTCT, the SNV ACTGCAGTCT, and the complex variant ACTGCAAGTCT[7]. For $k=4$, both variants share the 4-mer AGTC, but both take independent paths from the reference 4-mer CTGC to AGTC. In this case, we need to walk AGTC twice.

5 Discussion and Future Work

In this work, we have presented algorithms that allow for efficient sequence comparison using an *indexed de Bruijn* graph, and have then demonstrated how this can be used in a local reassembly pipeline. We are working to build a variant calling and genotyping platform using this algorithm, and hope to demonstrate results on real data soon.

One limitation of current local reassemblers is that they cannot simultaneously reassemble two regions of the reference genome. This strategy *may* be useful if we have two regions $\mathcal{R}_1$ and $\mathcal{R}_2$ with high similarity. Since a global aligner may not properly place reads into $\mathcal{R}_1$ or $\mathcal{R}_2$ during alignment, the reads aligned to $\mathcal{R}_1$ and $\mathcal{R}_2$ could be placed into the same pool $\mathcal{P}$ for reassembly. By emitting putative variants from this graph *and* using a genotyping model that incorporated phasing, this structure may be able to correct for global alignment errors between similar regions. Pooling multiple regions together can be used for detecting structural variants. If graph G constructed from $\mathcal{R}_1, \mathcal{R}_2, \mathcal{P}$ contains an arc $s_0 \to \cdots \to s_n$ where $s_i \in \mathcal{P}, \notin \mathcal{R}_1, \mathcal{R}_2, \forall i \in \{1, \ldots, n-1\}$ and $s_0 \in \mathcal{P}, \mathcal{R}_1, \notin \mathcal{R}_2, s_n \in \mathcal{P}, \mathcal{R}_2, \notin \mathcal{R}_1$, then the arc from s_1 to s_n identifies a possible structural variant breakpoint between $\mathcal{R}_1$ and $\mathcal{R}_2$.

Although this paper motivates the use of *indexed de Bruijn* graphs for variant calling, they can also be used for canonicalizing the output of multiple variant calling tools. Specifically, assume that we have two variants that are within n bases of each other. For these two variants, we can generate sequences $\mathcal{V}_1, \mathcal{V}_2$ by substituting the variant into $\mathcal{R}$ overlapping the site. We can then construct an *indexed de Bruijn* graph from $\mathcal{V}_1, \mathcal{V}_2$ with k-mer length $k \geq n$. If both $\mathcal{V}_1, \mathcal{V}_2$ overhang the variants by $\geq k$ bases and the graph contains a single path, the two variant representations are equivalent. This presents a more principled strategy for canonicalizing variants than the RESCUE strategy used in the SMASH benchmarking suite [13].

---

[7]At CCG, we have inserted an A between the two Cs, and changed the second C $\to$ A. Alternatively, this can be described as a deletion of the second C and an insertion of AA between CG.

5.1 Limits of Processing Repeated Sequences

In this paper, we constrain our sequences to not contain repeated subsequences of length greater than k, where k is the k-mer length used when constructing a *de Bruijn* graph G. If there is a repeated subsequence with length $\geq k$, then G will contain a cycle. Although *de novo* assembly methods attempt to resolve repeats through scaffolding [15], local reassembly methods frequently disallow repeats. Both the GATK's `HaplotypeCaller` [4] and `Scalpel` [10] increment the k-mer size until the repeat is eliminated or the tool gives up on locally reassembling the region.

5.2 Methods for Processing Spurs

Although local reassembly methods typically trim spurs from the haplotype reassembly graph to reduce the graph's complexity, the statistics from the bases in these spurs are assigned to locations on a haplotype during the realignment phase and are captured during genotyping/variant calling. Thus, in practice, we would like to place spurs in our graph as an small edit off of the reference genome or of an arc in $\mathcal{P}$. Note that a spur is created whenever a sequencing error occurs within k bases of the start or end of a read. Although we suggest exact realignment in §3, we expect errors in Illumina data to be single nucleotide errors, and thus searching for a single nucleotide substitution edit should suffice. If the spur branches off of a path in $\mathcal{R}$, then it is likely best to realign the spur to $\mathcal{R}$. If the spur branches off of a divergent variant within k bases of the variant either branching off of or rejoining $\mathcal{R}$, then we can realign the spur to the remainder of the variant's path, plus any necessary bases from $\mathcal{R}$.

6 Conclusion

In this paper, we have introduced the *indexed de Bruijn* graph. This extension of the traditional *de Bruijn* graph allows for $\Omega(n)$ local alignment of two or more sequences, with a hard $\mathcal{O}(n)$ bound for canonical edits and an $\mathcal{O}(l^2)$ bound for non-canonical edits (in genotyping, l is the variant allele length, which is typically much smaller than n). After describing this structure, we have demonstrated how *indexed de Bruijn* graphs can be used with a pooled model to locally reassemble haplotypes that contain variants. By using an *indexed de Bruijn* graph, we are able to reduce the runtime complexity of the local reassembly algorithms that are commonly used for variant discovery and calling.

References

[1] ALBERS, C. A., LUNTER, G., MACARTHUR, D. G., MCVEAN, G., OUWEHAND, W. H., AND DURBIN, R. Dindel: accurate indel calls from short-read data. *Genome research 21*, 6 (2011), 961–973.

[2] BAO, R., HUANG, L., ANDRADE, J., TAN, W., KIBBE, W. A., JIANG, H., AND FENG, G. Review of current methods, applications, and data management for the bioinformatics analysis of whole exome sequencing. *Cancer informatics 13*, Suppl 2 (2014), 67.

[3] BLONIARZ, A., TALWALKAR, A., TERHORST, J., JORDAN, M. I., PATTERSON, D., YU, B., AND SONG, Y. S. Changepoint analysis for efficient variant calling. In *Research in Computational Molecular Biology* (2014), Springer, pp. 20–34.

[4] DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., Hanna, M., et al. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nature genetics 43*, 5 (2011), 491–498.

[5] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge Univ Press, 1998.

[6] Fang, H., Wu, Y., Narzisi, G., O'Rawe, J. A., Barrón, L. T. J., Rosenbaum, J., Ronemus, M., Iossifov, I., Schatz, M. C., and Lyon, G. J. Reducing INDEL calling errors in whole genome and exome sequencing data. *Genome Med 6* (2014), 89.

[7] Garrison, E., and Marth, G. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907* (2012).

[8] Kingsford, C., Schatz, M. C., and Pop, M. Assembly complexity of prokaryotic genomes using short reads. *BMC bioinformatics 11*, 1 (2010), 21.

[9] Li, H. Towards better understanding of artifacts in variant calling from high-coverage samples. *arXiv preprint arXiv:1404.0929* (2014).

[10] Narzisi, G., O'Rawe, J. A., Iossifov, I., Fang, H., Lee, Y.-h., Wang, Z., Wu, Y., Lyon, G. J., Wigler, M., and Schatz, M. C. Accurate de novo and transmitted indel detection in exome-capture data using microassembly. *Nature methods 11*, 10 (2014), 1033–1036.

[11] Rimmer, A., Phan, H., Mathieson, I., Iqbal, Z., Twigg, S. R., Wilkie, A. O., McVean, G., Lunter, G., WGS500 Consortium, et al. Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics 46*, 8 (2014), 912–918.

[12] Smith, T. F., and Waterman, M. S. Identification of common molecular subsequences. *Journal of molecular biology 147*, 1 (1981), 195–197.

[13] Talwalkar, A., Liptrap, J., Newcomb, J., Hartl, C., Terhorst, J., Curtis, K., Bresler, M., Song, Y. S., Jordan, M. I., and Patterson, D. SMASH: A benchmarking toolkit for human genome variant calling. *Bioinformatics* (2014), btu345.

[14] Ukkonen, E. Algorithms for approximate string matching. *Information and control 64*, 1 (1985), 100–118.

[15] Zerbino, D. R., McEwen, G. K., Margulies, E. H., and Birney, E. Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. *PloS one 4*, 12 (2009), e8407.