Genome analysis

Unbiased pangenome graphs

Erik Garrison 1* and Andrea Guarracino 2

1Department of Genetics, Genomics and Informatics, University of Tennessee Health Science Center, Memphis, TN 38163, USA and 2Genomics Research Centre, Human Technopole, Viale Rita Levi-Montalcini 1, Milan 20157, Italy

*To whom correspondence should be addressed.

Abstract

Motivation: Pangenome variation graphs model the mutual alignment of collections of DNA sequences. A set of pairwise alignments implies a variation graph, but there are no scalable methods to generate such a graph from these alignments. Existing related approaches depend on a single reference, a specific ordering of genomes or a de Bruijn model based on a fixed k-mer length. A scalable, self-contained method to build pangenome graphs without such limitations would be a key step in pangenome construction and manipulation pipelines.

Results: We design the seqwish algorithm, which builds a variation graph from a set of sequences and alignments between them. We first transform the alignment set into an implicit interval tree. To build up the variation graph, we query this tree-based representation of the alignments to reduce transitive matches into single DNA segments in a sequence graph. By recording the mapping from input sequence to output graph, we can trace the original paths through this graph, yielding a pangenome variation graph. We present an implementation that operates in external memory, using disk-backed data structures and lock-free parallel methods to drive the core graph induction step. We demonstrate that our method scales to very large graph induction problems by applying it to build pangenome graphs for several species.

Availability and implementation: seqwish is published as free software under the MIT open source license. Source code and documentation are available at https://github.com/ekg/seqwish. seqwish can be installed via Bioconda https://bioconda.github.io/recipes/seqwish/README.html or GNU Guix https://github.com/egarris5/seqwish.scm.

Contact: egarris5@uthsc.edu

1 Introduction

A pangenome models the full genomic information of a species or clade (Garrison, 2019; Medini et al., 2005; Sherman and Salzberg, 2020). In contrast to reference-based approaches that relate sequences to a particular reference genome, methods that use pangenome reference systems attempt to model the mutual relationship between all represented genomes (The Computational Pan-Genomics Consortium, 2018). Many approaches model the pangenome alignment as a pangenome graph (Garrison et al., 2018; Hickey et al., 2020; Yokoyama et al., 2019). A pangenome graph encodes DNA sequences as walks through an underlying language encoded in a sequence graph (Hein, 1989). In a pangenome graph, variation can be understood in the context of any part of any included genome (Eizenga et al., 2020). This lets us avoid the problem of reference bias, which can be understood as the limitation of analyses to genome sequences that are similar to a chosen reference genome.

An unbiased pangenome graph would represent the alignment of all included genomes to all others. Existing methods approximate this relationship by progressive alignment to a graph initially based on a reference genome (Li et al., 2020), through a global structuring of the genome relationships in a neighbor joining phylogenetic tree (Armstrong et al., 2020), or via creation of a de Bruijn graph based on a fixed k-mer length (Minkin et al., 2016; Sheikhizadeh et al., 2016; Yu et al., 2021). These methods limit computational costs by reducing the number of pairwise comparisons, but in turn their results depend on input genome order, selected reference, guide-tree topology or k-mer length.

We consider the problem of building a pangenome graph without these potential sources of bias. Such a graph would be an ideal system to represent variation between two or more high-quality genomes. Given the rapid development of complete genome assemblies for humans and other vertebrates (Nurk et al., 2021; Rhie et al., 2021), we need a practical approach that can achieve this for tens to thousands of genomes on commodity hardware. Here, we present seqwish, an algorithm for the generation of a pangenome graph from pairwise alignments. Our solution is simple, but experiments on diverse sequence collections demonstrate that it easily scales to large pangenome building problems.
2 Algorithm

In this section, we provide a formal definition of variation graph induction. We then examine the bounds of a naive implementation of this algorithm. Finally, we propose compression and partitioning techniques to reduce the space and working memory complexity of the induction process by a large constant factor modulated by the degree of sequence divergence in the input pangenome. This yields a practical algorithm for variation graph induction that can scale to the largest available pangenomes.

2.1 Variation graph induction

**Definition 2.1.** Variation graphs are a common formalism to encode pangenome graphs (Garrison, 2019). In the variation graph \( G = (V, E, P) \), nodes (or vertices) \( V = v_1 \ldots v_{|V|} \) contain sequences of DNA. Each node \( v_i \) has a unique identifier \( i \) and an implicit reverse complement \( \overline{v_i} \). A node strand \( s \) corresponds to one node orientation. Edges \( E = e_1 \ldots e_{|E|} \) connect ordered pairs of node strands \( (e_i = (s_i, \overline{s}_i)) \), encoding the base topology of the graph. Paths \( P = p_1 \ldots p_{|P|} \) describe walks over node strands \( (p_i = s_1 \ldots s_{|p_i|}) \), representing the collection of genomes embedded in the graph.

**Theorem 2.1.** A variation graph represents pairwise alignments between its embedded paths.

**Proof.** By Definition 2.1, two paths have identical subsequences where they walk (or step) through the same series of oriented nodes (e.g. \( s_{12} s_{34} \)). An identical set of path steps is thus equivalent to a sequence match. Pairwise alignments are by definition collections of character-level matches between sequences. The variation graph thus models a set of pairwise alignments between paths in \( P \).

**Theorem 2.2.** We can build a variation graph from sequences and pairwise alignments. The resulting variation graph fully embeds both the sequences and all pairwise relationships in the input.

This follows from 2.1. Our input \( Q = S \cup \overline{S} \) is a set of \( N \) DNA sequences \( S = s_1 \ldots s_N \) and their reverse complements \( \overline{S} = \overline{s}_1 \ldots \overline{s}_N \). A match \( m = (i, j) \) asserts the aligned equivalence of two characters in sequences in \( Q \). Pairwise alignments between sequences in \( Q \) are a set of matches \( A = \{m_1 \ldots m_k\} \). By standard definition, each sequence matches its own reverse complement, that is \( g[i] = g[|g|-i] \) for all \( i = |g| - i \), and we assume these matches are included in \( A \). The transitive closure of a match, \( m^+ = \{i \ldots j\} \), is a set of characters in \( A \) that are transitiveley linked together by other matches. By definition of \( m \), each \( m^+ \) implies a single, identical character \( c(m^+) \).

We build a graph \( G \) inductively. We take the first match in \( A \), \( m_1 \), and execute a union-find operation to obtain \( m_1^+ \). We add the character of the match \( c(m_1^+) \) as a node \( v_1 \) in \( G \), and record the mapping from \( m_1^+ \rightarrow v_1 \). To induce the graph, we take the next unused match in \( A \): \( m_2 \notin \{v_1 \ldots v_{|V|}\}, \) obtain \( m_2^+ \) and add \( c(m_1^+) \) to \( G \). To allow the annotation of paths, we record the set of characters in \( Q \) that match to a given node in \( G \) in mapping \( Z = Q \rightarrow V = m_1 \ldots m_{|Q|} \). We continue until all matches have been used. Finally, we establish paths \( P \) by walking them in \( G \) using \( Z \), and record edges \( (E) \) where nodes occur successively in paths.

**Proof.** After the first step of induction, the graph represents all pairwise matches in \( m_1^+ \). Each subsequent step includes progressively more of \( A \), until at completion, all pairwise relationships are accounted for in \( V \).

The set of alignments represented by a variation graph may be strictly larger than the set of alignments used to induce it. The graph must contain at least the set of alignments given in input. In other words, by definition of \( G \), it cannot contain less match information than represented in the set of matches \( A \). However, the graph may also contain new implied pairwise relationships that arise due to transitive match relationships, as shown in Figure 1 for closures 1 and 6.

**2.2 Induction algorithm sketch**

For the sake of time and space complexity analysis, we consider a simple algorithm to implement the induction process. The induction depends on our ability to compute transitive closures of matches \( m^+ \). If \( A \) is sorted, we can find the matches of a given character in \( Q \) using binary search, which allows us to compute \( m^+ \) for each character. We do this non-redundantly by marking each used character in \( Q \) in an auxiliary data structure \( X \), which could be encoded as a bit-vector of length \( |Q| \). As we compute the matches \( m, m^+ \), we can sort \( Q \) and looking up their mapping in \( Z \) using binary search. The edge set \( E \) are the unique pairs of steps found in \( P \), and can be computed by sorting pairs of steps in \( P \).

**2.3 Naïve algorithm bounds**

The inductive proof of Theorem 2.2 demonstrates how to build a variation graph from sequences and their pairwise alignments. However, a naïve algorithm based on this model would require a very large amount of space. Although our identifier space \( |S| \) must include all of \( Q \), in practice, we only store \( X, Z, P \) can be trivially computed. Assume an all-to-all alignment of \( N \) sequences in \( A \) as input, and that all sequences are approximately identical, so that the induced variation graph has \( |S|/N \) nodes. The induction must maintain reference to all characters in all input sequences \( O(|S|) \), all character-to-character matches \( O(|A|) \approx O(|S|^2) \), the mapping of \( Q \) into the graph \( O(|Q|) \approx O(|S|) \), the nodes of the graph \( O(|V|) \approx O(|S|/N) \), the size of the edge set \( O(|E|) \approx O(|S|) \) and the set of paths \( O(|P|) \approx O(|S|) \). We also maintain the bitvector \( X \) to mark seen characters of \( Q \) during graph induction, which requires \( O(|S|) \) bits, equivalent to \( O(|S|/\log_2|S|) \) integer identifiers. In total, naïve Theorem 2.2-based induction would require approximately \( O(|S|^2 + 1/N + 1/\log_2|Q| + 4) \) space, or simply \( O(|S|^2) \).

Assuming that we want to build a graph of 100 haploid human genomes of \( 3 \times 10^9 \) bp, where \( N = 100 \) and \( |S| \approx 10^{11} \), we might
expect to use approximately $10^{15}$ identifiers to store the full model. Such a design is almost infeasible for inputs larger than a handful of genomes. For instance, we would need approximately $3 \times 10^{12}$ identifiers for just five human genomes. Although it is feasible to compute such a graph using external memory, the approximately 200-fold increase in space relative to the input renders this clearly impractical.

Considering the time complexity of induction, we anticipate $O(|A| \log |A|)$ time to sort the match set and $O(|\mathcal{V}| \log |A|)$ to query it and compute our $\mathcal{V}$ transitive closures. Computing the variation graph paths $\mathcal{P}$ involves converting sequences in $\mathcal{S}$ to walks through $\mathcal{V}$. We first sort the sequence-to-graph mapping array $\mathcal{Z}$ in $O(|\mathcal{S}| \log |\mathcal{S}|)$ operations, and then compute $\mathcal{P}$ in $O(|\mathcal{S}|)$ queries which each cost $O(|\mathcal{S}|)$. To obtain unique edges and generate $\mathcal{E}$, we must build and sort an array of size $O(2|\mathcal{P}|)$, and then iterate through it for $O(2|\mathcal{P}| \log 2|\mathcal{P}| + 2|\mathcal{P}|)$ operations. In sum, we would expect to require $O(|A| \log |A| + V \log |A| + 2|\mathcal{S}| \log |\mathcal{S}| + 2|\mathcal{P}| \log 2|\mathcal{P}| + |\mathcal{P}|)$. Using our approximate relationships to $|\mathcal{S}|$ given previously and simplifying, we arrive at $O(|\mathcal{S}|) 2N^2 \log N^2 / \log |\mathcal{S}| + (1/N + 4) \log |\mathcal{S}| + 2 + \log(4)$. Due to our dependence on sorting, and the logarithmic-time cost of queries, growth in $|\mathcal{S}|$ drives $O(|\mathcal{S}| \log |\mathcal{S}|)$ growth in overall complexity. As $N$ grows, both time and space complexity are dominated by the number of alignments, which in the case of our example is $O(N^2)$. For large numbers of highly similar genomes, we may not require all pairs of alignments to build a graph that contains all pairwise alignments. Various approaches could be used to reduce the size of $A$ without disrupting the induced graph. We leave these to later work.

### 2.4 Match compression

As the bounds analysis shows, space requirements make it impractical to apply a trivial version of Theorem 2.2 to generate a large pan-genome graph. Therefore, we need a compression approach that can reduce the redundancy in the input genomes to reduce the costs of the algorithm. When working with large numbers of genomes, alignments dominate the computational costs. A simple technique is to generalize matches $m = (i, j)$, which are between individual characters, to range-matches over pairs of ranges of characters in $Q$. For highly similar sequences, our expectation is that exact matches will occur in long runs. If the average pairwise diversity of sequences in our input is $1/k$, we expect exact matches to be around $k$ characters long. By encoding matches as pairs of ranges of characters, $r = (a, b) \cdot a, b = (i, j) \in Q$, we can obtain an approximately $k$-fold compression of $A$, yielding the range-match array $\mathcal{A}$. If sorted, $\mathcal{A}$ can be treated as an implicit interval tree (Li and Rong, 2021), which allows queries of containment and overlap in $O(|\mathcal{A}| \log |A|)$ time. This compression requires trivial changes to our graph induction model. To obtain our match transitive closures $(m^*)$, we query $\mathcal{A}$ for the range of a single character in $Q$, computing the character-level transitive relationships from the relative offsets of the ranges in $\mathcal{A}$. Match compression thus reduces our alignment storage memory bounds by a factor of $k$ without affecting our time complexity bounds.

This encoding can be used to replace the sequence-to-graph mapping $\mathcal{Z}$, yielding $\mathcal{Z}$. Rather than pairs of characters in $Q$ and $\mathcal{V}$, we record runs of matches between them as range matches. Although in expectation the length of these matches should be strictly less than $k$, due to the interruption of the graph by variation between genomes, this still allows us to reduce the size of $Z$ using runs of matches between $Q$ and $\mathcal{V}$. Additionally, we store the inverse of $\mathcal{Z}$, which maps ranges from $\mathcal{V} \rightarrow Q$, as $\mathcal{Z}$. We use $\mathcal{Z}$ to compact non-branching regions of $G$ into single nodes, and $\mathcal{Z}$ to accelerate results to appropriate data structures. This partitioning can introduce boundary effects which change the contents of $\mathcal{Z}$ and $\mathcal{Z}$ by splitting ranges at the boundaries of our partitions. However, while this will affect the compressed node definition $\mathcal{V}^*$, it does not affect $\mathcal{V}$, and it can be corrected via a post-processing step to sort and compact the id space.

### 2.5 Node compaction

For simplicity, we have thus far presented a character-level model of variation graph induction. However, range (or run) compression can also reduce the representation size of the graph. Rather than recording an identifier for each character in a sequence graph, it is useful to compact characters that form trivial linear components in the graph into single nodes. Broadly, the size of nodes will be bounded by the average distance between variants, which, for pan-genomes built from $\sim 100$ individuals of the same species, often provides a great reduction in the total number of nodes (and thus identifiers) required for $G$ and its components.

To compact $G$, we traverse $Q$, finding each entry in $\mathcal{Z}$ in turn, recording its start and end in $\mathcal{V}$, which can be understood as a character vector or string containing all the sequence in the nodes of $G$. We subsequently use these markings to subdivide $\mathcal{V}$ into a compacted version $\mathcal{V}'$ where compacted node boundaries are marked in an auxiliary bitvector $B : |B| = |\mathcal{V}|$ such that the first character in each compacted node is marked by a $1$ and other characters are marked $0$. $B$ allows us to compute compacted node ids using efficient rank operations (Gog et al., 2014).

### 2.6 Induction partitioning

Although match compression provides an approximate factor $k$ improvement in memory bounds for key data structures used in the induction, the approach we present in Section 2.4 requires working memory in the order of the set of transitive match closures in the graph. A simple approach to reduce this bound is to divide the induction problem into smaller pieces. We do so by computing the graph induction for a collection of initial characters in $Q$. In each partition, we apply a lock-free parallel union-find algorithm to derive the match closures (Anderson and Well, 1991), appending results to appropriate data structures. This partitioning can introduce boundary effects which change the contents of $\mathcal{Z}$ and $\mathcal{Z}$ by splitting ranges at the boundaries of our partitions. However, while this will affect the compressed node definition $\mathcal{V}^*$, it does not affect $\mathcal{V}$, and it can be corrected via a post-processing step to sort and compact the id space.

### 3 Implementation

We have presented a complete model for variation graph induction from sequences and their pairwise alignments (Algorithm 1). Here, we describe our specific implementation of this algorithm: seqwish. In general, our approach uses external memory to elaborate the graph, taking advantage of the availability of low-latency storage media, like solid-state drives, to maximize the performance of this approach.

### 3.1 Input and output processing

Our implementation reads standard data formats, FASTA or FASTQ for the input sequences, and PAF (Li, 2018) for pairwise alignments. It writes the graph in standard graphical fragment assembly (GFA) format (https://github.com/GFA-spec/GFA-spec).

In PAF, the input set of alignments is not directly expressed in terms of matches between specific characters in $Q$. Rather, each record lists the name of the aligned pair of query and target sequences and offsets in each. To efficiently process the input PAF, we thus need to build a sequence index that allows us to generate $A$. In particular, we build a compressed suffix array (Sadakane, 2000) over sequence names, that we call seqidx, and provide auxiliary supporting data structures that allow us to map between our input and the abstract concatenation of all input sequences and their reverse complements $(Q)$. We often build graphs from very large collections of sequences, such as raw sequencing reads or contigs from many thousands of samples. This seqidx avoids the overheads associated with a hash table on string names of input sequences. To enable highly efficient random access, we cache the input sequences in a disk-backed version of $Q$, into which our queries of sequence name and offset point. This trades time that might be spent accessing a compressed representation of the input for space in external memory.

For output in GFA, we iterate over nodes in $\mathcal{V}'$, writing each as a node record. Edges are similarly produced from the disk-backed multiset representing $\mathcal{S}$. The most computationally expensive part of graph emission is the rendering of the input sequences $S$ as paths $\mathcal{P}$ through the graph. For each input sequence in the seqidx, we walk
Algorithm 1: The seqwish graph induction algorithm. For the sake of simplicity, we omit the details of several query algorithms that interact with the input alignments, the transitive match closure, implicit interval tree construction and query, node generation and bitvector rank queries used in node compaction. Similarly, we omit the details of the input partitioning that we use to reduce maximum resident memory requirements.

```
Algorithm 1: The seqwish graph induction algorithm.

input: sequences $S$ and their alignment $A$
output: variation graph $G = (V, E, P)$

$A \leftarrow$ MakeMatchITree($A$) // alignment matches
$V \leftarrow \emptyset$/vector containing the set of nodes
$X \leftarrow$ BitVector($0, |S|$) // seen characters of $S$

// for each character in the input
for $i = 1$ to $|S|$ do

  // this character is not yet in $G$
  if $X_i = 0$ then

    // characters in $S$ matched to $i$
    $m_i \leftarrow$ GetTransitiveMatches($A, i$)

    // $V$ contains a node for each length $j$ interval in $G$
    $j = |V|$ // the node id or rank in $G$

    for $z \in m_i$ do

      // record last node boundary
      $X[i] \leftarrow 1$/mark seen character

      // extend ranges
      $Z \leftarrow$ ExtendRanges($Z, z, i$) // query -- graph
      $Z \leftarrow$ ExtendRanges($Z, j, z$) // graph -- query
    end
  end

// set up our $S \rightarrow V$ mappings
$Z \leftarrow$ MakeITree($Z$); $Z \leftarrow$ MakeITree($Z$)

// compact nodes in $V$ yielding $V'$
$V' \leftarrow \emptyset$; $I \leftarrow I$; $B = 0$; $B =$ BitVector($0, |V|$)

// for $i = 1$ to $|V'|$ do

$m = $ Overlaps ($Z, i$)

if $m \neq \emptyset$ then

  // $B[i]$ = 1/record a node boundary
  $V' \leftarrow$ AddNode($V', V'[b \ldots]$)

  $b = i$/record last node boundary

end

$I = m$/our last set of matching ranges

$P \leftarrow \emptyset$; $E \leftarrow \emptyset$/ paths and edges
$q = 1$/ for each sequence in the input

for $i = 1$ to $N$ do

  $p_i \leftarrow \emptyset$; $j = q$; $y = 0$

  // extend our path with the next step
  $(a, b) \leftarrow$ FirstOverlap($Z, j$)

  $x = $ NodeMatching($V', B, (a, b)$)

  $p_j \leftarrow p_j + x$/ extend the path

  $j \leftarrow j + (b - a)$/ increment offset in $S$

  $E \leftarrow E \cup ((y, x))$/ add to our edge set

  $y \leftarrow x$/ increment last step

  while $j < q + |g[j]|$

  $q \leftarrow \#$/ increment our pointer in $S$

end

return $G = (V', E, P)$
```

through the offsets in $S$ contained in the sequence and look up their mapping into $V'$ using $Z$. Range compression allows us to complete one lookup per range. By definition, each character in $S$ is covered by only one range in $Z$. We can thus iterate through the ranges in $Z$ without considering each character. Following the GFA format, we are able to independently generate $P$, as each path is represented on a separate record in the GFA.

3.2 Key disk-backed data structures

In our implementation, we rely on several basic external memory kernels. To reduce working memory requirements to an absolute minimum, we use a disk-backed version of the implicit interval tree that memory-maps the sorted array of intervals (https://github.com/ekg/mmmulti). Indexing the implicit interval tree requires a sorting step which dominates the runtime of our algorithm. We adapt the current best-performing in-place parallel sorting algorithm, In-place Parallel Super Scalar Sortslopes (IPS⁻⁰), to work on a disk-backed, memory-mapped array (Axtmann et al., 2017). This allows us work with $A$, $Z$ and $V'$ in external memory. By storing pairs of numerical identifiers in the backing array, we are able to generate a disk-backed multiset model which we use to compute the unique set of edges $E'$ in terms of offsets in $V$. The graph sequence vector $V'$ is simply written by appending characters to a file. We mark nodes to generate $V'$ using a bitvector kept in main memory, over which we subsequently generate a rank/select dictionary (Gog et al., 2014) for support of the final emission of the graph $G$.

3.3 Short match filter

Building a graph from an all-to-all alignment does not guarantee that the local structure of the graph is easy to understand. The all-to-all alignment is not coordinated, with each mapping aligned in isolation, and in consequence it fails to resolve the indel alignment normalization problem (Mose et al., 2019). This ambiguity can introduce deeply looping structures in the graph which collapse polymorphic microsatellites and other short VNTRs into very small numbers of nodes with very complex local topologies. Such motifs can cause problems with downstream analysis. We find that ambiguity about the arrangement of very short matches tends to drive complex local structures in the graph.

We mitigate this issue with a simple filter, seqwish-k, which simply ignores exact matches that are shorter than $k$ characters. This filter necessarily increases the size of the induced graph. But, it also replaces complex motifs shorter than $k$ with single bubbles. In doing so, it also removes short, expensive matches, reducing the overall space requirements for seqwish. When set very high, this filter can be used to generate a coarse, high-confidence graph built only from very long exact matches which will tend to be unique in the genome. Although the application of the $k >$ filter can result in a graph that is relatively ‘under-aligned’, we can further refine it through the application of local multiple sequence alignment (Gao et al., 2020) or graph normalization (https://github.com/marschall-lab/GAFFix). In a pangenomic context, underalignment caused by $k >$ match filtering can be mitigated by transitive relationships present in the pangenome.

4 Results

We evaluate seqwish through application to four pangenomes collected from Arabidopsis thaliana, Homo sapiens, Helicobacter pylori and Zea mays. This limited survey is intended to demonstrate basic scaling properties of the method, and its practicality when applied to real pangenomes. We also consider the effect of the minimum match length filter described in Section 3.3. Experiments were conducted on compute nodes with 386 GB of RAM and AMD EPYC 7402P processors with 48 vCPUs.

To construct the graph we first generate alignments with wmatch (https://github.com/waveygang/wmatch), a DNA sequence aligner
et al. (2018) with an extension of the wavefront algorithm (Marco-Sola et al., 2021) capable of obtaining base-level alignments for whole genomes. MashMap2 allows the user to define a homology length and pairwise divergence, expressed as a percent identity, over which to generate homology maps. This is useful when constructing pangenome graphs, because, in contrast to methods that are based on k-mer chaining (Harris, 2007; Li, 2018), it allows us to query the homology space of input genomes using two easily interpretable parameters. The version of wfmash used in these experiments allows us to align sequences with up to 10% divergence between them, providing highly sensitive input for our experiments. Our use of wfmash is pragmatic, and any mapping method capable of generating PAF with base-level alignments in CIGAR strings is capable of being used as input to seqwish.

In Table 1, we provide input and constructed graph parameters for a single parameter setting of wfmash and seqwish, obtaining graph statistics with ODGI (Guarracino et al., 2022). Figure 2 displays runtime versus graph size relative to the average input genome length across the range of parameters chosen for each pangenome. These provide a consistent set of insights. Reducing the sensitivity of alignments by increasing the identity threshold results in larger graphs. Filtering short matches results in larger graphs too, and for higher divergence collections of genomes, like H. pylori, tends to increase the size of the graph in Gbp and the number of its connected components.

Table 1. Performance of the graph induction algorithm

| Species       | Sequences | Haplotypes | fasta.Gbp | min.match.bp | time.seconds | memory.Gbytes | disk.Gbytes | graph.Gbp | Components |
|---------------|-----------|------------|-----------|--------------|--------------|---------------|-------------|-----------|------------|
| A.thaliana    | 922       | 16         | 1.90251   | 49           | 468          | 43.1287       | 7.1218      | 0.234284  | 100        |
| H.sapiens     | 17472     | 38         | 114.627   | 49           | 46598        | 347.4983      | 604.4261    | 4.47126   | 387        |
| H.pylori      | 292       | 250        | 0.407782  | 49           | 777          | 74.9484       | 20.2070     | 0.01421   | 5          |
| Z.mays        | 46289     | 41         | 90.2491   | 49           | 31043        | 351.1235      | 402.8716    | 13.8838   | 925        |

Notes: For each pangenome, we report a single experiment with seqwish -k filter set to 49 bp. From left to right, the columns indicate the species, the number of sequences (i.e. the number of contigs), number of haplotypes (i.e. the number of individuals), the sum of the length of all sequences in Gbp, the length of the short match filter applied in bp, the time in seconds and the amount of memory and disk space in Gbytes required for the graph induction, the length of the resulting graph in Gbp and the number of its connected components.

4.1 Comparison to related graph building methods

We have so far described our method for graph induction, provided detail on its implementation and demonstrated its performance over a set of pangenome graph building problems. These provide intuition about seqwish and its behavior. However, the reader may wonder how the method compares to other similar methods for graph construction, or what variation might be caused by changes in the alignment method used.

To focus on these questions, we examine the case of a single chromosome (chrV) of Saccharomyces cerevisiae for which we have seven assemblies (Yue et al., 2017). We compute 100 random permutations (Williams, 2009) of these assemblies and provide them to several graph construction methods, including minigraph (Li et al., 2020), TiaoPaCo (Minkin et al., 2016) and seqwish based on miniMap2 (Li, 2018) or wfmash (https://github.com/waveygang/wfmash) alignments.

Of existing methods, TiaoPaCo provides a graph in a form that is similar to seqwish’s. However, it is based on a de Bruijn graph that must have a relatively low k-mer length, leading to complex topologies that in our experience cannot be resolved easily by increasing k. Furthermore, the overlaps between nodes are incompatible with the variation graph model, and must be reduced or ‘bluntified’ with additional processing steps to be usable as input to variation graph tools (Eizenga et al., 2021). Because it is based on a de Bruijn graph, which is symmetrically constructed from a set of kmers and not the sequences themselves, we do not expect TiaoPaCo to be biased with respect to input genome order.

In contrast, minigraph develops a kind of partial order alignment (Lee et al., 2002) over the minimizers (sparse k-mer set) of input genomes, and it progressively builds the graph by adding new variation from each genome in order. We do expect it to be order and...
Although this particular study focuses on only a single small eu-
karyotic chromosome, it reveals that our basic understanding of ref-
ence bias in these graph construction methods is correct. By de-
design, progressive pangenome graph construction methods will be
affected by input genome order. But, other symmetric methods avoid order bias.

5 Discussion

We have presented a straightforward algorithm to generate a pange-
nome graph from a collection of genomes and alignments between them.
By exploiting a simple model of this algorithm, we provide com-
putational bounds that give insight into the complexity of the
problem. We then make this approach practical by applying the con-
cept of match compression, which reduces the expected computa-
tional complexity by a factor proportional to the diversity of input
sequences. Our experimental results demonstrate that we can apply
our method to various collections of sequences and alignments. It
easily scales to some of the largest species pangenome construction
problems possible using publicly available, high-quality genome
assemblies. seqwish is a generic sequence graph inducer of potential-
ly many uses. We envision that it can serve as a component in di-
verse sequence analysis and assembly pipelines, and hope that our
thorough description of its core algorithm and functionality will en-
able its reuse by other researchers.

It is also a potentially novel approach. Despite the existence of
many methods for pangenome building, we are not aware of any
comparable method which can losslessly convert an all-to-all align-
ment to a variation graph. This direct relationship allows users to
adjust the shape of the resulting graph by modifying alignment
parameters, allowing the design of custom graph construction pro-
cesses based on domain-specific knowledge and potentially manual
curation of assembly alignments. In contrast to existing methods,
which depend on particular structuring of their input (Armstrong
et al., 2020; Li et al., 2020), seqwish is unbiased in that as it directly
and uniformly represents sequence relationships given on input in the
resulting graph. Our comparison with related pangenome graph-
building methods demonstrates that these effects can be significant.

As we observe in Figure 3, minigraph indeed generates a distribu-
tion of graph lengths with clusters corresponding to permutations
beginning with the same genome. This suggests that most of the vari-
ation in minigraph’s graph derives from the first genome which is
picked. The graphs are thus biased toward the chosen base refer-
ance (first input) biased, although the degree of bias requires ex-
perimentation to establish.

Finally, seqwish itself is by definition unbiased with respect to in-
put order. But, if the alignments are affected by changes in order,
the resulting graph will also change.

We use ODGI to compute the size, node count and edge count of
the resulting graphs (Guarracino et al., 2022). Changes in these sta-

tistics indicate changes in graphs. It also gives an indication of the
scale of the differences, which may indicate how large an effect of a
different construction approach might be. If graph parameters are
the same for two large graphs, we consider them equivalent, which
is not formally correct but provides a sufficient approximation for
this analysis.

As we observe in Figure 3, minigraph indeed generates a distribu-
tion of graph lengths with clusters corresponding to permutations
beginning with the same genome. This suggests that most of the vari-
ation in minigraph’s graph derives from the first genome which is
picked. The graphs are thus biased toward the chosen base refer-
ance. However, this is not the case for the seqwish builds based on
minimap2 and wfmash, which appear to have almost exactly the
same length across all permutations.

In Table 2, we further consider variance in the length, node
count and edge count of all graphs. This shows that, as expected,
minigraph’s output depends markedly on input genome order. We
also find that the minigraph+seqwish configuration is not order-

Table 2. Variance in graph properties across genome input order permutations

| Method            | μ bp    | σ bp    | μ node | σ node | μ edge | σ edge |
|-------------------|--------|--------|--------|--------|--------|--------|
| minigraph         | 648 967| 8112   | 190    | 11.6   | 267    | 15.9   |
| seqwish.mm2       | 609 039| 4.08   | 30 692 | 2.89   | 41 723 | 6.32   |
| seqwish.wfm       | 683 470| 28 756 | 0      | 0      | 39 080 | 0      |
| twopaco.k19       | 1 270 115| 26 383 | 0      | 0      | 35 284 | 0      |

Notes: We report the mean (μ) and variance in standard deviations (σ) for graph length (μ bp), node count (μ node) and edge count (μ edge). We show results from minigraph, TwoPaCo with k=19 (twopaco.k19), seqwish using minimap2 alignments (seqwish.mm2) and seqwish using wfmash alignments (seqwish.wfm).

The results indicate that TwoPaCo is a powerful tool for generating pangenome graphs. It produces graphs with a smaller length variance and a more uniform node and edge count compared to other methods.

In conclusion, we have presented a new method for generating pangenome graphs that is unbiased by input order and allows users to adjust the shape of the resulting graph through alignment parameter modification. This method, seqwish, can be used as a component in diverse sequence analysis and assembly pipelines, providing a powerful tool for researchers.

References:

- Garrison, E., & Guarracino, A. (2022). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
- Armstrong, M. M., et al. (2020). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
- Li, H., et al. (2020). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
- Minkin, M., et al. (2016). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
- Sheikhzadeh, E., et al. (2016). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
- Yu, E., et al. (2021). Seqwish: A generic sequence graph inducer with applications to pangenome graph building. bioinformatics, 38(17), 3304-3306.
and pangenome graph construction modality are high, they are in some sense more manageable. The modular, independent nature of the hardest part of the computation—the derivation of the alignments—can be scaled out to large compute clusters as well as cached for incremental construction and update of large pangenomes.

We furthermore expect that local giant components will tend to arise in the alignment graph without requiring a complete pairwise alignment, suggesting that lessons from graph theory may help to guide a justified sparsification, or downsampling, of the input (Janson et al., 1993). This suggests that simple random sampling of alignments may evoke minimal changes in graph structure, provided that there are many homologous copies of each locus in the pangenome. It will also be possible to apply seqwish to measure at what level of mapping sparsification we observe changes in the induced graph. Although this may be infeasible for very large problems, it will allow us to develop an empirical understanding of how to reduce the complexity of the initial alignment step without affecting the resulting pangenome graph.

Acknowledgements

We are grateful to members of the Human Genome Structural Variation Consortium (HGSVC) and Human Pangenome Reference Consortium (HPRC) production teams for their development of resources used in our exposition. We thank the authors of the pangenome resources made available on GenBank which have made our experiments possible.

Funding

We gratefully acknowledge support from NIH/NIDA U01DA047638 and NSF PivoSS Award #2118709 (E.G.) and efforts by Dr. Nicole Soranzo to establish a pangenome research unit at the Human Technopole in Milan, Italy.

Data availability

Code and links to data resources used to build this manuscript and its figures can be found in the paper’s public repository: https://github.com/pangenome/seqwish-paper.

Conflict of Interest: none declared.

References

Anderson, R.J. and Woll, H. (1991) Wait-free parallel algorithms for the union-find problem. In: STOC ’91: Proceedings of the twenty-third annual ACM symposium on Theory of Computing January 1991, pp. 370–380. https://doi.org/10.1145/103418.103458

Armstrong, J. et al. (2020) Progressive cactus is a multiple-genome aligner for the thousand-genome era. Nature, 587, 246–251.

Astmann, M. et al. (2017) In-place parallel super scalar samplesort (IPSSSSo). In: 25th Annual European Symposium on Algorithms (ESA 2017), Vol. 87 of Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 9:1–9:14.

Eizenga, J. et al. (2020) Pangenome graphs. Annu. Rev. Genomics Hum. Genet., 21, 139–162.

Eizenga, J. et al. (2021) Walk-preserving transformation of overlapped sequence graphs into blunt sequence graphs with GetBlunted. In: De Mol, L. et al. (eds.), Connecting with Computability. Springer International Publishing, pp. 169–177.

Gao, Y. et al. (2020) abPOA: an SIMD-based C library for fast partial order alignment using adaptive band. Bioinformatics.

Garrison, E. (2019) Graphical pangenomics. PhD thesis, Cambridge University.

Garrison, E. et al. (2018) Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat. Biotechnol., 36, 875–879.

Garrison, E. et al. (2022) pggb: the PanGenome graph builder. https://github.com/pangenome/pggb

Gog, S. et al. (2014) From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J. and Katajainen, J. (eds.), Experimental Algorithms, Springer International Publishing, pp. 326–337.

Guarracino, A. et al. (2022) ODGI: understanding pangenome graphs. Bioinformatics, 38, 3319–3326.

Harris, R.S. (2007) Improved pairwise alignment of genomic DNA. Pennsylvania State University, USA.

Hein, J. (1989) A new method that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when the phylogeny is given. Mol. Biol. Evol., 6, 649–668.

Hickey, G. et al. (2020) Genotyping structural variants in pangenome graphs using the vg toolkit. Genome Biol., 21, 35.

Jain, C. et al. (2018) A fast adaptive algorithm for computing whole-genome homology maps. Bioinformatics, 34, i748–i756.

Janson, S. et al. (1993) The birth of the giant component. Random Struct. Algor., 4, 233–358.

Liao, W.-W. et al. (2020) Multiple sequence alignment using partial order graphs. Bioinformatics, 36, 452–464.

Li, H. (2018) Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics, 34, 3094–3100.

Li, H. and Rong, J. (2021) Bedtk: finding interval overlap with implicit interval tree. Bioinformatics, 37, 1315–1316.

Li, H. et al. (2020) The design and construction of reference pangenome graphs with minigraph. Genome Biol., 21, 265.

Liao, W.-W. et al. (2022) A draft human pangenome reference. Cold Spring Harbor Laboratory. https://www.biorxiv.org/content/early/2022/07/09/2022.07.09.499321

Marino-Sola, S. et al. (2021) Fast gap-affine pairwise alignment using the wavefront algorithm. Bioinformatics, 37, 436–463.

Medini, D. et al. (2005) The microbial pan-genome. Curr. Opin. Genet. Dev., 15, 589–594.

Minin, N. et al. (2016) Twopaco: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. Bioinformatics, brw069, page

Mose, L.E. et al. (2019) Improved indel detection in DNA and RNA via re-alignment with abra2. Bioinformatics, 35, 2966–2973.

Nurk, S. et al. (2021) The complete sequence of a human genome. BioRxiv.

Rhee, A. et al. (2021) Towards complete and error-free genome assemblies of all vertebrate species. Nature, 592, 735–746.

Sadakane, K. (2000) Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Goos, G. et al. (eds.), Algorithms and Computation, Springer Berlin Heidelberg, pp. 410–421.

Sheikhzadeh, S. et al. (2016) PanTools: representation, storage and exploration of pan-genomic data. Bioinformatics, 32, i487–i493.

Sherman, R.M. and Salzberg, S.L. (2020) Pan-genomics in the human genome era. Nat. Rev. Genet., 21, 243–254.

The Computational Pan-Genomics Consortium (2018) Computational pan-genomics: status, promises and challenges. Brief Bioinform., 19, 118–135.

Williams, A. (2009) Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In: Proceedings of the 2009 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, pp. 987–996.

Yokoyama, T.T. et al. (2019) MoMi-G: modular multi-scale integrated genome graph browser. BMC Bioinformatics, 20, 548.

Yu, C. et al. (2021) Stliter: a novel algorithm to iteratively build the compacted de Bruijn graph from many complete genomes. IEEE/ACM Trans. Comput. Biol. Bioinform., 19, 2471–2483.

Yue, J.-X. et al. (2017) Contrasting evolutionary genome dynamics between domesticated and wild yeasts. Nat. Genet., 49, 913–924.