AlfaPang: Alignment Free Algorithm for Pangenome Graph Construction

Adam Cicherski
Institute of Informatics, University of Warsaw, Poland

Anna Lisiecka
Institute of Informatics, University of Warsaw, Poland

Norbert Dojer
Institute of Informatics, University of Warsaw, Poland

Abstract

The success of pangenome-based approaches to genomics analysis depends largely on the existence of efficient methods for constructing pangenome graphs that are applicable to large genome collections.

In the current paper we present AlfaPang, a new pangenome graph building algorithm. AlfaPang is based on a novel alignment-free approach that allows to construct pangenome graphs using significantly less computational resources than state-of-the-art tools. The code of AlfaPang is freely available at `https://github.com/AdamCicherski/AlfaPang`.

2012 ACM Subject Classification Applied computing → Computational genomics

Keywords and phrases pangenome, variation graph, genome alignment, population genomics

Digital Object Identifier 10.4230/LIPIcs.WABI.2024.23

Supplementary Material *Software (Source code)*: `https://github.com/AdamCicherski/AlfaPang` [4], archived at `swh:1:dir:e8a27a620673d796d0701ab29a39aa2383bece22`

Funding This work was supported by the National Science Centre, Poland, under grant number 2022/47/B/ST6/03154.

1 Introduction

Pangenome (or variation) graphs serve as models for joint representation of populations of genomes [20, 6, 9, 2, 10]. They have proven to be useful in analyzing sequence evolution and variation [3, 14], as well as in reducing the so-called reference bias in the analysis of experimental data [13, 21].

However, the success of the pangenome-based approaches depends on the existence of efficient construction methods, applicable to large collections of genomes. Most pangenome building algorithms adapt the approaches used in whole genome alignment tools. Early versions of the `VG` toolkit [13] constructed pangenome graphs iteratively, i.e. aligning consecutive sequences to a current graph. In the current version of `VG`, by default, graphs are constructed from genomic sequences using `Minigraph-Cactus` [15], which aligns all genomes to a reference genome.

In both approaches the outcome depends on an arbitrary choice of genome order (`VG`) or the reference (`Minigraph-Cactus`). To avoid such biases, several alternatives have recently been proposed. `seqwish` [11] builds pangenome graphs from all-to-all pairwise genome alignments. Unfortunately, the construction doesn't scale linearly with respect to the number of genomes, and the final graph requires refinement. The last problem was addressed in `pggb` [12] – a pipeline that builds a pangenome graph in three steps:

1. all-to-all genome alignment (`wfmash`),
2. graph induction from pairwise alignment (`seqwish`),
3. graph refinement (`smoothxg+gfaffix`).

All the above-mentioned tools construct variation graphs, which are the most widely used, but not the only, graph pangenome models. One of the most important alternatives are de Bruijn graphs. Their structure is strictly determined by the parameter k, which guarantees the avoidance of order and reference biases. Moreover, the construction is conceptually simple, and optimized building algorithms such as `TwoPaCo` [18] or `bifrost` [16] are orders of magnitude faster than alignment-based building algorithms for variation graphs. However, de Bruijn graphs pose a challenge for downstream analysis, especially in terms of annotation, visualization and information extraction [1].

A bridge between both models that could combine their advantages was proposed in [5]. The paper introduces the notion of a string graph, which is a common generalization of variation graph and de Bruijn graph. Moreover, the authors propose an axiomatization of the desired properties of representing a sequence collection in such a graph. It is shown that the axioms are always satisfied in de Bruijn graphs and that they determine the structure of variation graphs up to merging unbranched paths into single nodes and the opposite operation. Furthermore, authors explore the relationship between de Bruijn graphs and variation graphs satisfying the axioms to design an algorithm transforming the former into the latter. The proposed transformation algorithm can potentially be used as a crucial component of an efficient variation graph building pipeline.

In the current work, we take this path to achieve an efficient variation graph construction one step further. We design and implement `AlfaPang` – an algorithm that builds a variation graph satisfying the axioms introduced in [5] directly from the input sequences. We show that replacing the first two steps of `pggb` with our algorithm results in significant efficiency improvement and yields output graphs of similar properties. The rest of the paper is organized as follows. In section 2 we introduce the necessary notation and prove theoretical results underlying the correctness of our algorithm. In section 3 we describe the algorithm and study its complexity. Sections 4 and 5 present experiments' design and results, respectively.

2 Representing sequences with variation graphs

2.1 Directed variation graphs

A *directed variation graph* is a tuple $G = \langle V, E, l \rangle$, where:

- V is a set of vertices,
- $E \subseteq V^2$ is a set of directed edges,
- $l : V \to \Sigma^+$ is a function labeling vertices with non-empty strings over the DNA alphabet $\Sigma = \{A, C, G, T\}$.

A *path* in a variation graph is a sequence of vertices $\langle v_1, \dots, v_m \rangle$ such that $\langle v_j, v_{j+1} \rangle \in E$ for every $j \in \{1, \dots, m-1\}$. The set of all paths in G will be denoted by $\mathcal{P}(G)$. The labeling function l extends to $\hat{l} : \mathcal{P}(G) \to \Sigma^+$ defined by formula $\hat{l}(\langle v_0, \dots, v_m \rangle) = l(v_0) \cdot \dots \cdot l(v_m)$, i.e. the label of the path is the concatenation of the labels of its consecutive vertices.

Assume that $G = \langle V, E, l \rangle$ is a directed variation graph and $\sim$ is an equivalence relation on the set of G-nodes satisfying $v \sim v' \Rightarrow l(v) = l(v')$ for all $v, v' \in V$. The quotient graph of G by $\sim$ is defined as $G' = \langle V', E', l' \rangle$, where

- $V' = V/\sim$,
- $E' = \{\langle [v]_\sim, [v']_\sim \rangle \mid \langle v, v' \rangle \in E\}$,
- $l'([v]_\sim) = l(v)$.

The correctness of the definition of l' (i.e. independence on the choice of $[v]_\sim$-representative) is guaranteed by the above assumption on $\sim$. Note that the quotient construction saves the labels of paths, i.e. given a path $p = \langle v_1, \ldots, v_m \rangle$ in G, $p' = \langle [v_1]_\sim, \ldots, [v_m]_\sim \rangle$ is a path in G' and $\hat{l}'(p') = \hat{l}(p)$.

Given a path $p = \langle v_1, \ldots, v_m \rangle$, the *subpath* of p is any path $p[j_1..j_2] = \langle v_{j_1}, \ldots, v_{j_2} \rangle$, where $1 \le j_1 \le j_2 \le m$. Similarly, $S[j_1..j_2]$ denotes the substring of a string S consisting on the characters from positions $j_1, \ldots, j_2$.

When $|l(v)| = 1$ for every vertex v, the graph is called *singular*. In this case $|p| = |\hat{l}(p)|$ and moreover $\hat{l}(p[j_1..j_2]) = \hat{l}(p)[j_1..j_2]$ for all $1 \le j_1 \le j_2 \le |p|$. To simplify the description, we assume below that considered variation graphs are singular. This does not lead to a loss of generality, because any variation graph can be transformed into a singular one by splitting each node v into a path with $|l(v)|$ nodes. We refer readers interested in generalized definitions to [5].

2.2 Representations of collections of sequences

Given a set of sequences $\mathcal{S} = \{S_1, \ldots, S_n\}$, a singular directed variation graph $G\langle V, E, l\rangle$ and $\pi : \mathcal{S} \to \mathcal{P}(G)$, we say that $\langle G, \pi \rangle$ *represents* $\mathcal{S}$ iff the following conditions are satisfied:

- $\hat{l}(\pi(S_i)) = S_i$ for every $i \in \{1, \ldots, n\}$,
- every vertex in G occurs in some path $\pi(S_i)$,
- every edge in G joins two consecutive vertices in some path $\pi(S_i)$.

We define the set of *positions* in S as $Pos(S) = \{\langle i, j \rangle \mid 1 \le i \le n \wedge 1 \le j \le |S_i|\}$. The set of *$\pi$-occurrences* of a vertex v is defined as $Occ_\pi(v) = \{\langle i, j \rangle \in Pos(S) \mid \pi(S_i)[j] = v\}$. Let's call the *generic representation* of $\mathcal{S}$ the representation $\langle G_0, \pi_0 \rangle$, where $G_0 = \langle V_0, E_0, l_0 \rangle$ and:

- $V_0 = Pos(S)$,
- $E_0 = \{\langle \langle i, j \rangle, \langle i, j+1 \rangle \rangle \mid 1 \le i \le n \wedge 1 \le j < |S_i|\}$,
- $l_0(\langle i, j \rangle) = S_i[j]$,
- $\pi_0(S_i) = \langle \langle i, 1 \rangle, \ldots, \langle i, |S_i| \rangle \rangle$.

▶ **Lemma 1.** *For every singular representation $\langle G, \pi \rangle$ of $\mathcal{S} = \{S_1, \ldots, S_n\}$, there exists an equivalence relation $\sim_{\langle G,\pi \rangle}$ on $Pos(S)$ such that:*

1. *G is isomorphic to a quotient graph of G_0 by $\sim_{\langle G,\pi \rangle}$,*
2. *$\pi(S_i) = \langle [\langle i, 1 \rangle]_{\sim_{\langle G,\pi \rangle}}, \ldots, [\langle i, |S_i| \rangle]_{\sim_{\langle G,\pi \rangle}} \rangle$ for all $i \in \{1, \ldots, n\}$.*

Proof. The relation $\sim_{\langle G,\pi \rangle}$ is defined as follows:

$$\langle i, j \rangle \sim_{\langle G,\pi \rangle} \langle i', j' \rangle \iff \pi(S_i)[j] = \pi(S_{i'})[j']$$

Verifying that both conditions are satisfied is straightforward. ◀

Let $S_i, S_{i'}$ be two (not necessarily different) sequences from $\mathcal{S}$ and assume that they have a common k-mer $S_i[p..p+k-1] = S_{i'}[p'..p'+k-1]$. We say that π *reflects* this common k-mer iff it is represented by the same path in the graph, i.e. $\pi(S_i)[p..p+k-1] = \pi(S_{i'})[p'..p'+k-1]$. We say that $\langle G, \pi \rangle$ represents $\mathcal{S}$ *k-completely* iff all common k-mers in $\mathcal{S}$ are reflected by π.

We say the pair of π-occurrences $\langle i, j \rangle, \langle i', j' \rangle$ of a vertex v is:

- *directly k-extendable* iff these occurrences extend to a common k-mer reflected by π, i.e. $\pi(S_i)[j-m..j+m'] = \pi(S_{i'})[j'-m..j'+m']$ for some $m, m' \ge 0$ satisfying $m+m' \ge k-1$,
- *k-extendable* if there is a sequence of occurrences of v that starts from $\langle i, j \rangle$, ends at $\langle i', j' \rangle$ and each two consecutive occurrences in that sequence are directly k-extendable.

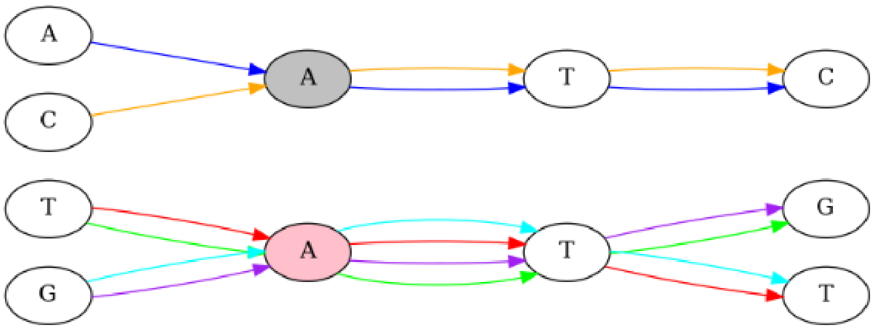


Figure 1 Example of a 3-faithful and 3-complete variation graph. Edge colors are used to mark genomic paths (graph has no multi-edges, they are used only for purpose of paths visualisation). All occurrences of vertex "A" filled with pink are 3-extendable, as occurrences on the red and green paths can both be extended to the path labeled with TAT, on the red and cyan paths to ATT, on the purple and cyan paths to GAT, and on the purple and green paths to ATG. On the other hand, occurrences of the grey vertex "A" on the orange and blue paths can be extended to ATC but are not extendable to any of the previously mentioned 3-mers, and therefore this vertex cannot be merged with the pink one.

We say that $\langle G, \pi \rangle$ represents $\mathcal{S}$ *k-faithfully* if every pair of occurrences of a vertex is k-extendable.

Note that the k-completeness property specifies, which fragments of $\mathcal{S}$-strings must be unified in the representation, while k-faithfulness states that anything that is not a consequence of k-completeness cannot be unified (see example on Figure 1).

▶ **Theorem 2.** *Let $\mathcal{S} = \{S_1, \dots, S_n\}$ be a set of sequences. Then the k-complete and k-faithful representation of $\mathcal{S}$ exists and is unique up to isomorphism.*

The above theorem is roughly equivalent to Theorems 1 and 2 in [5]. In that paper the proof of the existence was based on the transformation of a de Bruijn graph into a variation graph. Here we propose an alternative proof that leads to a more efficient variation graph construction algorithm.

Proof. Let $G_0 = \langle V_0, E_0, l_0 \rangle$ be a generic representation of $\mathcal{S}$. We define a binary relation $\sim_0$ indicating the pairs of positions in $\mathcal{S}$ that should be merged in a representation reflecting common k-mers:

$$\langle i, j \rangle \sim_0 \langle i', j' \rangle \iff \exists_{0 \le m < k}\ S_i[j-m..j+k-1-m] = S_{i'}[j'-m..j'+k-1-m]$$

Let $\sim$ denote the equivalence closure of $\sim_0$. Obviously, the above definition implies that $S_i[j] = S_{i'}[j']$ whenever $\langle i, j \rangle \sim_0 \langle i', j' \rangle$ and, consequently, the same property holds for $\sim$. Therefore a quotient graph G of G_0 by $\sim$ is properly defined. Moreover, each G_0-path $\pi_0(S_i) = \langle v_1, \dots, v_{|S_i|} \rangle$ can be handled to G through the quotient construction: $\pi(S_i) = \langle [v_1]_\sim, \dots, [v_{|S_i|}]_\sim \rangle$.

Hence we have a representation $\langle G, \pi \rangle$ of $\mathcal{S}$ that is:

- k-complete, because consecutive vertices in paths representing common k-mers were merged in the quotient construction,
- k-faithful, because all occurrences of a node v are in relation $\sim$, so for each pair $\langle i, j \rangle, \langle i', j' \rangle \in Occ_\pi(v)$ there exists a sequence $\langle i, j \rangle = \langle i_0, j_0 \rangle, \dots, \langle i_p, j_p \rangle = \langle i', j' \rangle$ of v-occurrences such that for each $l \in \{1, \dots, p\}$ the condition $\langle i_{l-1}, j_{l-1} \rangle \sim_0 \langle i_l, j_l \rangle$ is satisfied, which means that occurrences $\langle i_{l-1}, j_{l-1} \rangle$ and $\langle i_l, j_l \rangle$ are directly k-extendable.

Moreover, by Lemma 1, every representation $\langle G', \pi' \rangle$ of $\mathcal{S}$ is isomorphic to a quotient of G_0 by some relation $\sim'$ on the set of oriented G_0-vertices. It is easily seen that $\langle G', \pi' \rangle$ is

- k-complete iff $\langle i, j \rangle \sim \langle i', j' \rangle \Rightarrow \langle i, j \rangle \sim' \langle i', j' \rangle$,
- k-faithful iff $\langle i, j \rangle \sim' \langle i', j' \rangle \Rightarrow \langle i, j \rangle \sim \langle i', j' \rangle$.

Therefore, if $\langle G', \pi' \rangle$ has both properties, graphs G and G' must be isomorphic. ◀

2.3 Bidirected variation graphs

In bidirected graphs each node has two sides (denoted here ± 1) and undirected edges join incident nodes on particular sides [8]. Both sides are equivalent in the sense that swapping sides at selected nodes yields an isomorphic bidirected graph.

A path entering a node on one side must exit it on the other side. More formally, a *path* in a bidirected graph is a sequence $\langle \langle v_1, o_1 \rangle, \ldots, \langle v_m, o_m \rangle \rangle$ such that for all respective j

- $o_j = \pm 1$ determines the *orientation* of v_j in the path,
- side o_{j-1} of v_{j-1} is connected by an edge with side $-o_j$ of v_j.

Paths may be reversed, but it requires reversing both order and orientation of the nodes, i.e. given path $p = \langle \langle v_0, o_0 \rangle, \ldots, \langle v_m, o_m \rangle \rangle$, its reverse is $p^{-1} = \langle \langle v_m, -o_m \rangle, \ldots, \langle v_0, -o_0 \rangle \rangle$.

Bidirected variation graphs naturally represent the double-stranded structure of DNA. The orientation of a node indicates the strand of the represented DNA fragment, i.e. strand $\langle v, +1 \rangle$ has sequence $l(v)$, while $\langle v, -1 \rangle$ has sequence $l(v)^{-1}$, where S^{-1} denotes the reverse complement of sequence S. For convenience, we introduce notation $S^{+1} = S$ and $p^{+1} = p$. The label of the path is the concatenation of the oriented labels of consecutive vertices, i.e. $\hat{l}(\langle \langle v_0, o_0 \rangle, \ldots, \langle v_m, o_m \rangle \rangle) = l(v_0)^{o_0} \cdot \ldots \cdot l(v_m)^{o_m}$. Note that hence $\hat{l}(p^{-1}) = \hat{l}(p)^{-1}$.

Bidirected representations of sequence collections are defined similarly to directed ones. Positions and vertex occurrences are extended to include the orientation, i.e.

- $Pos(\mathcal{S}) = \{ \langle i, j, o \rangle \mid 1 \leq i \leq n \wedge 1 \leq j \leq |S_i| \wedge o = \pm 1 \}$,
- $Occ_\pi(v) = \{ \langle i, j, o \rangle \in Pos(\mathcal{S}) \mid \pi(S_i)[j] = \langle v, o \rangle \}$.

Because the strands of the represented sequences are treated in the same way, the concept of reflecting common k-mers applies to occurrences of k-mers on both strands. Below we adapt the definitions of k-completeness and k-faithfulness to take this into account.

Let $S_i, S_{i'}$ be two (not necessarily different) strings from $\mathcal{S}$ and assume that they have a common k-mer $S_i^o[p..p+k-1] = S_{i'}^{o'}[p'..p'+k-1]$ (o and o' indicate the strands, on which the k-mer occurs). We say that π *reflects* this common k-mer iff it is represented by the same path in the graph, i.e. $\pi(S_i)^o[p..p+k-1] = \pi(S_{i'})^{o'}[p'..p'+k-1]$. We say that $\langle G, \pi \rangle$ represents $\mathcal{S}$ *k-completely* iff all common k-mers in $\mathcal{S}$ are reflected by π.

We say the pair of π-occurrences $\langle i, j, o \rangle, \langle i', j', o' \rangle$ of a vertex v is:

- *directly k-extendable* iff these occurrences extend to a common k-mer reflected by π, i.e. $\pi(S_i)^o[j-m..j+m'] = \pi(S_{i'})^{o'}[j'-m..j'+m']$ for $m, m' \geq 0$ satisfying $m + m' \geq k-1$,
- *k-extendable* if there is a sequence of occurrences of v that starts from $\langle i, j, o \rangle$, ends at $\langle i', j', o' \rangle$ and each two consecutive occurrences in that sequence are directly k-extendable.

We say that $\langle G, \pi \rangle$ represents $\mathcal{S}$ *k-faithfully* if every pair of occurrences of a vertex is k-extendable.

▶ **Theorem 3.** *Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of DNA sequences. Then the k-complete and k-faithful representation of $\mathcal{S}$ as a singular bidirected variation graph exists and is unique up to isomorphism.*

Proof. As in the directed case, the desired representation is constructed as a quotient of the generic representation. The quotient construction is slightly more complicated in the bidirected case, because each node of the original graph can either retain or reverse its orientation in the resulting graph. However, the structure of the entire proof is analogous to the proof of Theorem 2, so we leave the details to the reader. ◀

3 AlfaPang algorithm

In this section we present `AlfaPang` – ALignment Free Algorithm for PANGenome graph construction. `AlfaPang` builds k-complete and k-faithful variation graphs following the quotient construction described in the proofs of Theorems 2 and 3.

3.1 Algorithm overview

Given a collection of sequences S and a positive natural number k, we first build its generic representation $G = \langle V, E \rangle$. Then we build a weighted bipartite graph with parts V and B, where B is a set of vertices labeled by canonical k-mers of S, and edges satisfy the following conditions:

- each edge e is assigned a value from the set $\{-k, \dots, -1, 1, \dots, k\}$, denoted as $C(e)$,
- $C(\langle\langle i, j\rangle, b\rangle) = c$ iff
  - $S_i[j - c + 1..j + k - c] = l(b)$ for $c > 0$,
  - $S_i[j - c - k..j - c - 1] = l(b)^{-1}$ for $c < 0$.

Therefore, an edge between $\langle i, j\rangle$ and b indicates that the position in the sequence corresponding to $\langle i, j\rangle$ can be extended to a k-mer represented by b, and the value assigned to the edge indicates its position in that k-mer. Hence such graph allows us to represent the relation described in previous section.

To find all vertices in G that should be merged with a chosen vertex v, we traverse the bipartite graph starting from v using a BFS manner, but with the following constraints:

- If we enter a vertex belonging to V, we can leave it by any edge.
- If we visit a vertex belonging to B from an edge with value c, we can leave it only through edges with value c or $-c$.

All vertices of V visited during one such run establish one equivalence class of the relation presented in the theorems. For each such class, we choose a canonical orientation arbitrarily to be consistent with a canonical label of the first vertex visited in the run. Therefore, to find a quotient graph, we start a new run as long as there are vertices not visited in previous runs.

3.2 Compact graph representation

We reduce the memory requirements of our algorithm by representing the redundant information from the bipartite graph implicitly. First, we store only the edges with values 1 or $-k$ and calculate the rest on the fly. This optimization is based on the observation that if $e_1 = \{\langle i, j\rangle, b\}$ and $C(e_1) = 1$, then for $1 < q < k$ and $e_2 = \{\langle i, j - q\rangle, b\}$, we have $C(e_2) = 1 + q$. Similarly, if $e_1 = \{\langle i, j\rangle, b\}$ and $C(e_1) = -k$, then for $1 < q < k$ and $e_2 = \{\langle i, j - q\rangle, b\}$, we have $C(e_2) = -k + q$. We can then modify the constraints for graph traversal:

- From a node belonging to an equivalence class, we can exit by an edge or jump $q < k$ positions backward in the generic representation and then traverse through the edge incident with that vertex (this vertex is not marked as visited since it does not need to belong to the same class).
- From vertices belonging to B, we can exit by both types of edges (those assigned to 1 and those assigned to $-k$).
- If we exit from a node belonging to B by an edge with the same sign as the edge we used to enter that vertex, we need to jump q vertices forward in the generic representation to find a vertex belonging to the equivalence class.
- If we exit from a vertex belonging to B by an edge with the opposite sign to the one we used to enter that vertex, we need to jump $k - 1 - q$ vertices forward to find a vertex belonging to the equivalence class.

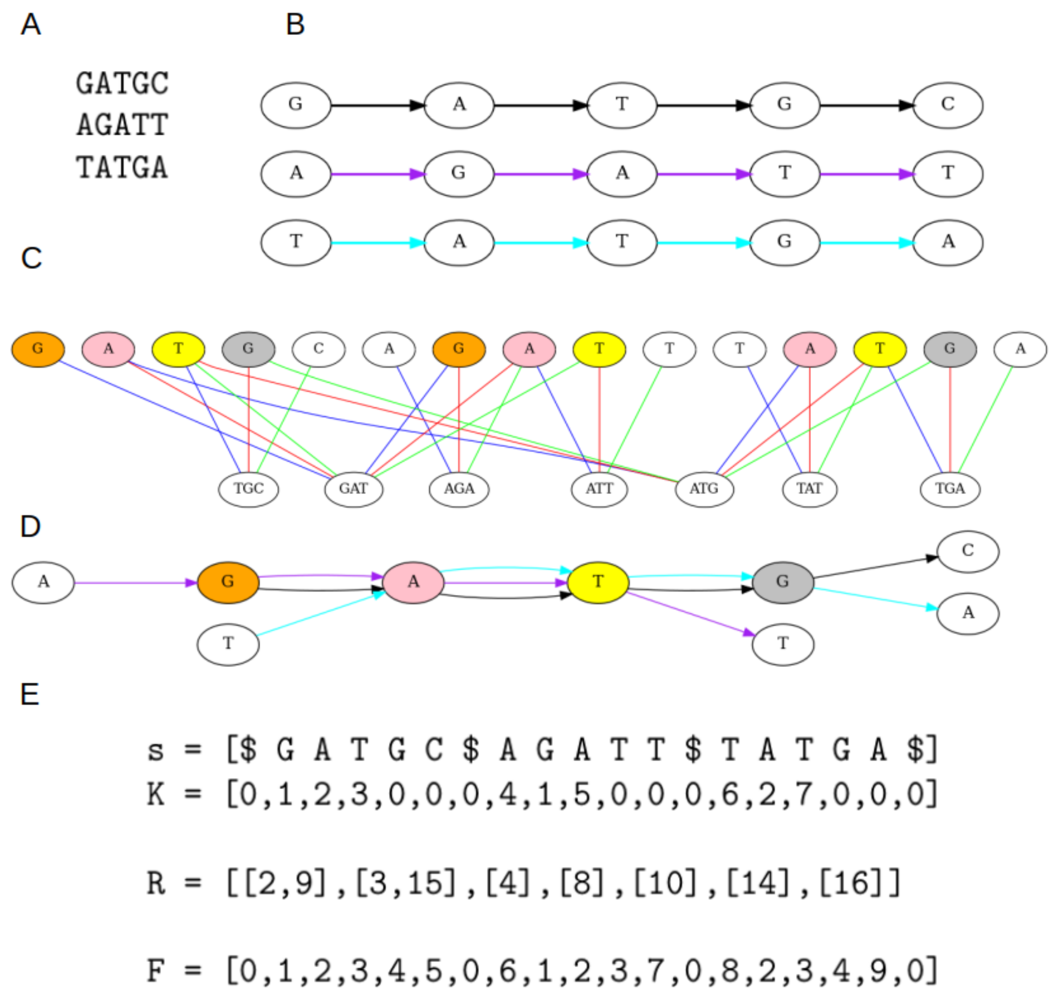
To implement the algorithm, we do not build such a graph explicitly. Instead, we use the following data structures:

- A concatenation of all sequences from the set S as a single string s, with an additional special character \$ at the ends of original sequences.
- A vector K of the same size as s, that stores the ID (positive integer) assigned to the k-mer b at position i if $S[i..i+k-1] = b$, or the negative of that value if $S[i..i+k-1] = b^{-1}$. If the k-mer starting at position i is not fully included in a sequence from S, then $K[i] = 0$. IDs of all canonical k-mers of S are obtained by enumeration with positive numbers, which can be done efficiently using a hash table.
- A vector R that stores M vectors, where M is the number of distinct canonical k-mers in S. This vector is used as an inverted index, enabling fast locating of k-mer occurrences in s. If for some $i, j > 0$ we have $K[i] = \pm j$, then we push i into the vector $R[j]$.
- A vector F of the same size as s, initially filled with zeros. This vector represents the assignment of positions in s to the vertices of the output graph. The absolute value of a number in this vector identifies a vertex ID, and its sign determines the vertex orientation.

In such framework we can identify vertices belonging to V with indices of K and vertices belonging to B with indices of R. Pseudocode showing the algorithm is presented in Listing 1. Figure 2 illustrates the data structures and a part of an example run of the algorithm.

3.3 Unbranched paths compactness

The graph resulting from the described algorithm is singular. As a final step in our tool, we compact unbranching paths into vertices to reduce its size. Since it is k-complete, we know that, (similarly to the vertices of the de Bruijn graph) each vertex has no more than $|\Sigma|$ vertices connected to it on each side, and each of these has a different symbol as a label. Therefore, based on ideas proposed in the [17], we implemented a similar algorithm for our variation graphs. For each vertex, we can assign a state that encodes whether the vertex is inside such a path or is a branching node. Initially, vertices are assigned to one of the $(\Sigma + 1)^2$ states representing symbols on their sides (+1 for a special character representing the end of a sequence), or to one of three special states representing branching vertices - two different states for situations when a vertex has only one edge on one side and more than one edge on the other, and a state for vertices with more than one edge on each side. To recognize branching vertices, we build a vector D of length equal to the number of vertices in the singular graph, filled with 0. Then we traverse vector F and update D as follows:



```
s = [$ G A T G C $ A G A T T $ T A T G A $]
K = [0,1,2,3,0,0,0,4,1,5,0,0,0,6,2,7,0,0,0]

R = [[2,9],[3,15],[4],[8],[10],[14],[16]]

F = [0,1,2,3,4,5,0,6,1,2,3,7,0,8,2,3,4,9,0]
```

Figure 2 **A**: Set of DNA sequences. **B**: Generic variation graph representation of that set. **C**: Bipartite graph for $k = 3$. Colors of edges represent values assigned to them (blue: 1, red: 2, green: 3). Nodes filled with a color other than white belong to the same equivalence class. Orange nodes are both connected to the 3-mer GAT by blue edges. The first and the second pink nodes are connected to GAT by red edges, and the first and third pink nodes are connected to ATG by blue edges. The first and the second yellow nodes are connected to GAT by green edges, and the first and the third are connected to ATG by red edges. Grey nodes are connected to ATG by green edges. **D**: Variation graph resulting from the quotient algorithm. Different edge colors mark different genomic paths (not multi-edges) and are consistent with edge colors in B. Node colors are consistent with equivalence classes shown in C. **E**: Data structures used in the algorithm: s – concatenation of the input sequences, K – vector storing ids of k-mers starting at given positions in s, R – inverted index, enabling locating of k-mer occurrences in s, F – output vector, assigning positions in s to the vertices of the output graph.
To find a pink equivalence class, let's start from symbol "A" at position 3 - we assign $F[3]$ a new value (2 in this example). Since $K[3] = 2$, we look into the vector $R[2]$, which points to positions 3 (starting position) and 15, so we assign $F[15] = 2$. Next, we backtrack one position to $K[2]$. Since $K[2] = 1$, we look into $R[1]$, which points to positions 2 (from where we came) and 9. From position 9, we move one position forward and assign $F[10] = 2$. After backtracking two steps from the starting position, we find $K[1] = 0$, indicating that this "A" is not the third symbol in any k-mer. We repeat the procedure from found positions 10 and 15, identifying no additional positions.

Listing 1 Quotient algorithm.

```
function quotient(s, K, R, F)
  v = 1.
  for i from 1 to |s|
    if F[i] != 0
      continue
    visited_edges = empty set
    queue = empty queue
    queue.push(i)
    while queue is not empty
      node_id = queue.pop()
      for j from 0 to k-1
        idx = K[node_id - j]
        if (idx, j) in visited_edges
          continue
        visited_edges.add((idx, j))
        for pos in R[idx]
          x = K[pos]
          new_node = pos + (idx != x) * (k - 1) + sign(idx * x) * j
          if F[new_node] != 0
            continue
          F[new_node] = v if s[new_node] == s[i] else -v
          queue.push(new_node)
    v = v + 1
  return F
```

- If $D[abs(F[i])] = 0$, we assign it a state encoding labels of the previous and next vertex ($s[i-1]$ and $s[i+1]$).
- Otherwise, we check if labels of vertices on both sides of a vertex are consistent with the state assigned to it. If so, we do nothing. Otherwise, we change the state to one of the special states (depending on which side of the vertex differs).
- If a vertex is assigned to a special state, we need to check only one of the sides or none of them.
- Additionally, we check if $-F[i-1] = F[i]$ or $F[i] = -F[i+1]$. If so, we also need to change the state to one of the specials.

Next, we simplify states by changing all non-special states to a single value. Then, we traverse vector F once again to merge maximal unbranching paths into vertices, identified by combinations of branching nodes at the ends of the paths.

3.4 Algorithm complexity

If we denote the sum of lengths of sequences in S as N, then the time complexity of our algorithm is $O(kN)$, since we need to traverse each edge in the bipartite graph exactly one. However, we only need $O(N)$ space to store vectors K, R, F, and the string s.

3.5 Code availability

C++ implementation of `AlfaPang` can be found at: `https://github.com/AdamCicherski/AlfaPang`

4 Design of experiments

4.1 Datasets and parameter setting

We tested our tool on two series of genome collections: the *Escherichia coli* series containing 50, 100, 200, and 400 *Escherichia coli* haplotypes, obtained from [12], which we further extended by datasets of 800, 1600, and 3412 sequences downloaded from GenBank (with total lengths ranging from $250Mbp$ to $17Gbp$; specific queries are available in our GitHub repository), and the *Saccharomyces cerevisiae* series containing 16, 32, 64, and 118 haplotypes (total length from $195Mbp$ to $1.44Gbp$), obtained from [19].

Before the analysis, we estimated the complexity and repetitiveness of the sequences in the datasets to get some intuition about the appropriate selection of the parameter k for `AlfaPang`. For this purpose, we calculated the fractions of *overrepresented* and *rare* k-mers, defined as the k-mers that occur more frequently than the number of genomes and that occur only once, respectively. Too low k yields a large fraction of overrepresented k-mers, which will result in merging non-homologous fragments. On the other hand, too large k increases the fraction of rare k-mers, which will translate into poor sequence similarity detection.

The fractions were calculated for k between 17 and 67 with step 10 in the datasets with 64 *S. cerevisiae* haplotypes and with 100 *E. coli* haplotypes. Results are shown in Table 1. Based on that, we decided to use $k = 47$ for all `AlfaPang` tests, since this value allows us to keep the fraction of overrepresented k-mers below 5% while having about twice as small the fraction of rare k-mers.

4.2 Compared algorithms and criteria

Similarly to `seqwish`, `AlfaPang` may produce pangenome graphs with complex local structures. Therefore we evaluate `AlfaPang` on two levels.

On the first level, we compared the computational efficiency of `AlfaPang` and the first two steps of the `pggb` workflow, i.e., `wfmash+seqwish`. Unfortunately, we were unable to complete the `wfmash+seqwish` calculations even for the 400 *E. coli* sequences – after 6 days of computations on a server with 512 GB RAM, the process crashed due to excessive memory consumption. Consequently, we repeated the calculations on the *E. coli* series with the Erdős–Rényi random graph sparsification option activated. With that approach, we were able to finish calculations for 400 and 800 sequences, but 1600 and 3412 still required too much memory.

In order to prepare the second-level comparison, we modified the `pggb` pipeline by replacing the first two steps with `AlfaPang`. Thus, the new workflow, called `AlfaPang+`, consists of `AlfaPang` followed by the graph refinement tools `smoothxg` and `gfaffix`. For all *Saccharomyces cerevisiae* datasets and datasets of 50, 100, 200, and 400 *E. coli* sequences, we

Table 1 Fractions of overrepresented and rare k-mers for different k.

kmer size	k=17	k=27	0k=37	k=47	k=57	k=67
S. cerevisia 64						
over represented k-mers	0.105	0.071	0.057	0.049	0.043	0.038
rare k-mers	0.011	0.018	0.024	0.030	0.037	0.043
E. coli 100						
over represented k-mers	0.109	0.076	0.059	0.049	0.042	0.037
rare k-mers	0.011	0.014	0.017	0.020	0.023	0.027

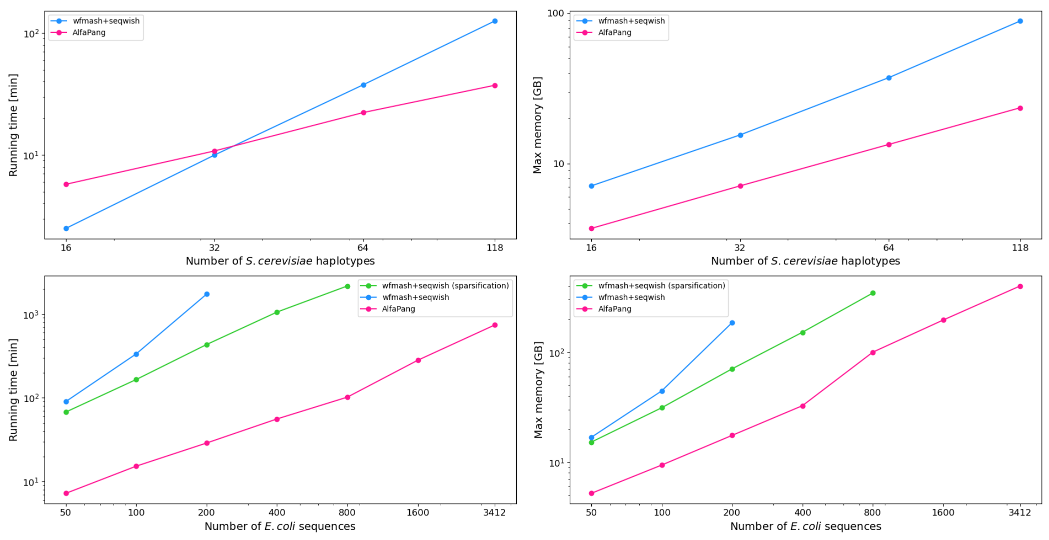


Figure 3 A log–log plots of performance of `AlfaPang` vs `wfmash+seqwish`. All `wfmash+seqwish` tests were performed using 20 CPU threads.

compared `AlfaPang+` with `pggb` and `Minigraph-Cactus` in terms of computational efficiency and output graph properties. Finally, we analyzed the similarity between sequence alignments induced by these graphs.

To define a measure of this similarity, we introduce the following equivalence relation: a pair of positions in two (not necessary different) sequences is *aligned* in graph G if both are represented by the same position in a label of the same vertex. In the case of singular graphs this could be expressed $S_i[p] \equiv_G S_j[p'] \iff \pi(S_i)[p] = \pi(S_j)[p']$. We define $\mu(G)$ as the set of all aligned pairs in G. The similarity between two variation graphs G_1, G_2 is measured using Jaccard index between $\mu(G_1)$ and $\mu(G_2)$.

Due to the large size of the sets of aligned pairs in our datasets, exact calculation of their intersections is computationally inefficient. Therefore we decided to estimate their cardinality using the mafTools package [7]. To do so we first converted variation graphs to alignments in Multiple Alignment Format (transforming each node into a separate MAF block), then calculated the estimates of ratios $\beta_1 := \frac{|\mu(G_1)\cap\mu(G_2)|}{|\mu(G_1)|}$ and $\beta_2 := \frac{|\mu(G_1)\cap\mu(G_2)|}{|\mu(G_2)|}$ using the mafComparator program from mafTools, and finally we used $\frac{\beta_1|\mu(G_1)|+\beta_2|\mu(G_2)|}{2}$ as an approximation of the cardinality of $\mu(G_1) \cap \mu(G_2)$.

5 Results

5.1 Performance comparison

The performance of each tool was evaluated in terms of running time and peak memory consumption. Running time was measured as wall clock time and peak memory as maximum resident set size using the `time` command. All benchmarks were performed on a Supermicro X10DRi server with 512GB RAM and two 14-cores CPUs Intel Xeon E5-2690V4, using single thread for AlfaPang and 20 threads for all other tools.

Figure 3 presents the comparison of `AlfaPang` and `wfmash+seqwish`, without the refinement step of `pggb`, on a log-log plots. `AlfaPang` consumes less memory than `wfmash+seqwish` on all datasets and is substantially faster than `wfmash+seqwish` on all *E. coli* datasets and

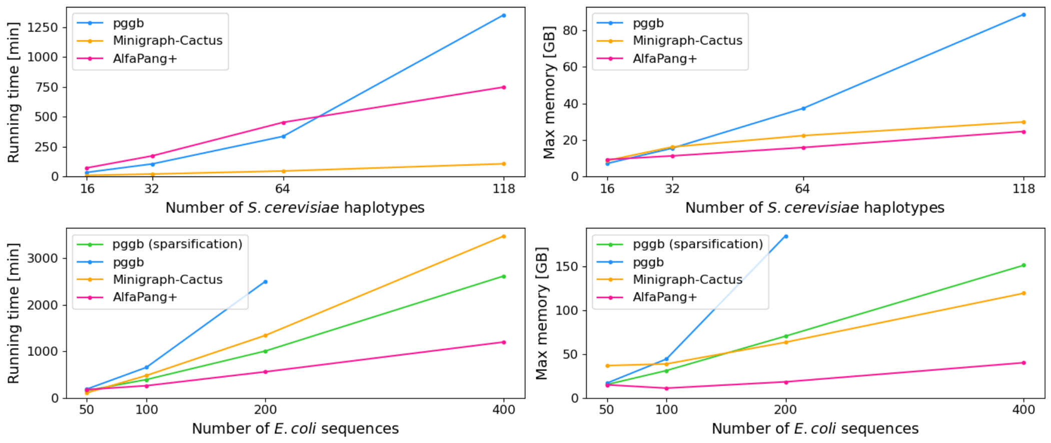


Figure 4 Performance of `AlfaPang+` vs `pggb` vs `Minigraph-Cactus`.
All `pggb` and `Minigraph-Cactus` tests, as well as `smoothxg` in `AlfaPang+` were performed using 20 CPU threads.

on the larger *S. cerevisiae* datasets. Moreover, unlike `wfmash+seqwish`, `AlfaPang` scales almost linearly with respect to the number of genomes (although there is a memory usage increase when scaling from 400 to 800 genomes due to the need to switch from 4-byte integers to 8-byte integers, but it resumes linear scaling afterward). Consequently, the difference between both tools increases with the dataset size. For example, on 50 *E. coli* sequences, `AlfaPang` is more than 9 times faster than `wfmash+seqwish` with sparsification (and 12 times faster than `wfmash+seqwish` without it) and consumes 3 times less memory, while on 800 *E. coli* sequences, `AlfaPang` is more than 21 times faster and consumes 3.5 times less memory. It is worth pointing out that `AlfaPang` achieved these results using only one thread, while `wfmash+seqwish` was tested on 20 CPU threads.

Figure 4 summarizes the computational efficiency of full pangenome building pipelines. For the series of *E. coli* datasets, `AlfaPang+` proved to be the most efficient tool and shows almost linear scalability with respect to the size of the data. On the dataset consisting of 400 *E. coli* sequences, `AlfaPang+` is more than twice as fast as `pggb` (with activated sparsification) and `Minigraph-Cactus`, and it consumes less than one-third of the memory required by the other tools. Although `Minigraph-Cactus` is faster on the smallest dataset of 50 *E. coli* sequences, `AlfaPang+` is still consuming significantly less memory.

For the series of *S. cerevisiae* datasets, the results are slightly more ambiguous. The runtime for both `pggb` and `AlfaPang+` is mostly dependent on `smoothxg`, which runs faster in `pggb`. However, as the number of haplotypes increases, `AlfaPang+` gains an advantage and becomes almost twice as fast for the largest dataset. Although `Minigraph-Cactus` proved to be the fastest tool in this case (3 to 13 times faster than `pggb`, depending on number of haplotypes, and 7 times faster than `AlfaPang+`), this comes at the expense of loosing interchromosomal structural variants in the resulting variation graph. As opposite to `pggb` and `AlfaPang+`, for all yeast datasets `Minigraph-Cactus` constructed graphs with 16 connected components, corresponding to 16 chromosomes of *S. cerevisiae*. This can be explained by the way the last tool is designed. After initial graph construction, `Minigraph-Cactus` remaps all genomic sequences onto the graph and assigns each sequence to one of the reference chromosomes. Sequences that cannot be confidently assigned to any chromosome are left out of further analysis. Thus, subsequent steps may be performed independently and in parallel for each reference chromosome, resulting in significant computational speedup for multichromosomal genomes, likely explaining the difference in the running time results of experiments on the *E. coli* and *S. cerevisiae* datasets.

As the number of *S. cerevisiae* haplotypes goes from 16 to 118, the memory usage increases by 12.5 times for `pggb`, 3.4 times for `Minigraph-Cactus`, and only 2.7 times for `AlfaPang+` which again demonstrates `AlfaPang+` scalability in this regard.

5.2 Graphs topology

To measure the complexity of the produced pangenome graphs, we compared such graph characteristics as the number of nodes and edges. Results are displayed in Table 2 and Table 3. We also compared the total length of nodes labels to check the ability of all the tools to compress input sequences (see Table 4). In all the following tables, we refer to the graphs produced by the `pggb` pipeline without sparsification, except for the graph of the largest *E. coli* dataset, for which sparsification was activated.

Table 2 Number of nodes (in 10^6).

	`AlfaPang+`	`pggb`	`Minigraph-Cactus`
S. cerevisia 16	0.92	0.89	1.08
S. cerevisia 32	1.78	1.80	2.22
S. cerevisia 64	2.88	2.85	3.86
S. cerevisia 118	3.73	3.87	5.44
E. coli 50	1.74	1.62	1.93
E. coli 100	1.93	1.79	2.67
E. coli 200	2.79	2.61	4.13
E. coli 400	3.66	3.25	5.80

Table 3 Number of edges (in 10^6).

	`AlfaPang+`	`pggb`	`Minigraph-Cactus`
S. cerevisia 16	1.27	1.22	1.48
S. cerevisia 32	2.47	2.48	3.05
S. cerevisia 64	4.07	3.98	5.37
S. cerevisia 118	5.40	5.49	7.68
E. coli 50	2.37	2.20	2.64
E. coli 100	2.65	2.45	3.65
E. coli 200	3.85	3.60	5.72
E. coli 400	5.09	4.53	8.16

For all *S. cerevisia* datasets, the number of nodes in graphs from `AlfaPang+` and `pggb` differs by at most 5%. For *E. coli* datasets these numbers are slightly higher (up to 11% for 400 haplotypes and 6-7% for the rest of datasets). `Minigraph-Cactus` produces graphs with visibly larger number of nodes and edges. On *S. cerevisia* datasets, `Minigraph-Cactus` graphs has 17-45% more nodes then the `AlfaPang+` (and 20-40% more than `pggb`), while on *E. coli* data it is 10-58% more than `AlfaPang+` (19-78% more than `pggb`).

For all graphs, the number of edges is 35-45% larger than the number of nodes. Moreover, going from 16 to 118 *S. cerevisia* haplotypes increases the number of nodes 4-5 times, while going from 50 to 400 *E. coli* sequences, the number of nodes increases by at most 3 times across all tools, indicating that for all tools graph size grows sublinearly with respect to the number of input sequences. Similar conclusion can be drawn from Table 4. When the number of *S. cerevisia* haplotypes increases from 16 to 118, the total size of node sequences increases by 1.3, 1.7, and 2.9 times for `AlfaPang+`, `pggb`, and `Minigraph-Cactus`, respectively. For *E. coli* data, increasing the number of sequences from 50 to 400 results in the total size of node sequences increasing by 1.8 and 3.1 times for `AlfaPang+` and `Minigraph-Cactus`, respectively. For `pggb`, the graph constructed from a dataset of 400 sequences deviates from this trend: increasing the number of sequences from 50 to 200 increases the total length of node labels by 1.4 times, while increasing from 200 to 400 sequences increases this value by 2.5 times. This deviation is suspected to result from the sparsification used for that dataset. For all datasets, `AlfaPang+` shows a higher rate of compression than `pggb`, which in turn has a higher rate of compression than `Minigraph-Cactus`.

5.3 Graphs similarity

Table 5 summarizes the number of aligned pairs in all graphs. It was consequently larger in `AlfaPang+` than in `pggb` which in turn was greater than in `Minigraph-Cactus` for all datasets. For *S. cerevisia* datasets, `pggb` identified 69-75% as many pairs as `AlfaPang+` and `Minigraph-Cactus` detected only 59-61% of the number of pairs found by `AlfaPang+`. For *E. coli* datasets, `pggb` found 68-92% as many pairs as `AlfaPang+` while `Minigraph-Cactus` found 59-85%. In both cases the percentage decreases as the number of sequences grows. As expected, number of aligned pairs scale quadratically with a number of sequences.

The Jaccard index between sets of aligned pairs is presented in Table 6. On the *S. cerevisia* datasets, the Jaccard index between `AlfaPang+` and `pggb` is approximately 70%, while for the pair `AlfaPang+` and `Minigraph-Cactus`, it is around 60%. The Jaccard index for `pggb` and `Minigraph-Cactus` ranges between 82% and 87% on *S. cerevisia* datasets, with a decreasing tendency as the number of haplotypes grows.

For *E. coli* datasets, the Jaccard index between `AlfaPang+` and `pggb` decreases from 89% to 65% as the number of sequences increases. A similar trend is observed for the pair `AlfaPang+` and `Minigraph-Cactus`, with the index decreasing from 83% to 57%. In contrast, the Jaccard index between `pggb` and `Minigraph-Cactus` varies between 74 and 91%, showing no clear dependencies on the number of sequences.

These numbers are close to the ratio of the numbers of aligned pairs between the tools. A more precise analysis of these values showed that `AlfaPang+` was able to find 96-99% of the pairs found by `pggb`. Moreover both `AlfaPang+` and `pggb` were able to found more than 97.9% and 98.5 % of the pairs found by `Minigraph-Cactus`, respectively. Thus, the differences between pair sets are mainly due to the differences in sensitivity to sequence similarity between tools.

Table 4 Total number of base pairs in nodes (in 10^6 bp).

Dataset	Alfapang+	pggb	mg	Input size
3 S. cerevisia 16	12.61	17.03	18.59	192.04
S. cerevisia 32	13.64	20.73	26.54	384.52
S. cerevisia 64	15.32	26.52	38.58	769.61
S. cerevisia 118	16.74	29.56	53.67	1416.23
E. coli 50	14.02	19.01	28.89	249.52
E. coli 100	14.91	20.20	43.61	539.83
E. coli 200	19.64	27.37	67.28	1050.67
E. coli 400	25.43	68.21	90.75	2027.21

Table 5 Number of aligned positions pairs (in 10^9). A pair of positions from input sequences is aligned in a graph if they are both represented by the same position in a label of the same vertex.

	`AlfaPang+`	`pggb`	`Minigraph-Cactus`
S. cerevisia 16	2.2	1.52	1.26
S. cerevisia 32	8.47	6.33	5.14
S. cerevisia 64	34.76	25.81	21.00
S. cerevisia 118	121	90.95	71.50
E. coli50	5.55	5.08	4.72
E. coli 100	28.51	25.31	19.04
E. coli 200	107.76	92.37	73.70
E. coli 400	503.56	343.97	298.79

Table 6 The estimated values of the Jaccard index between sets of aligned pairs. The numbers in parentheses are the estimated ratios of the number of common aligned pairs in both graphs to the numbers of aligned pairs found by the first and second tool, respectively.

	`pggb` vs `AlfaPang+`	`Minigraph-Cactus` vs `AlfaPang+`	`Minigraph-Cactus` vs `pggb`
S. cerevisia 16	68.3 (99.1, 68.7)	60.6 (99.4, 61.7)	86.6 (99.3, 87.5)
S. cerevisia 32	73.9 (98.5, 73.7)	63.5 (99.2, 64.6)	84.7 (99.2, 85.7)
S. cerevisia 64	70.8 (97.3, 72.2)	62.6 (99.0, 63.8)	84.3 (99.0, 85.3)
S. cerevisia 118	69.7 (95.7, 71.9)	61.3 (98.9, 62.5)	81.8 (99.0, 82.9)
E. coli 50	88.5 (98.3, 89.9)	82.7 (98.5, 83.8)	91.3 (99.1, 92.1)
E. coli 100	82.7 (96.3, 85.5)	65.6 (98.7, 66.0)	74.3 (99.1, 74.6)
E. coli 200	79.7 (96.1, 82.4)	66.4 (98.2, 67.1)	78.3 (98.9, 78.9)
E. coli 400	65.4 (97.5, 66.5)	57.3 (97.9, 58.0)	84.5 (98.5, 85.6)

6 Conclusion

We presented `AlfaPang` – a novel algorithm for building pangenome graphs. Unlike alternative algorithms, `AlfaPang` constructs graphs with their structure strictly defined by the k-completeness and k-faithfulness properties introduced in [5]. `AlfaPang`'s runtime and memory usage scales linearly with the number of genomes, allowing it to process much larger sets of genomes than state-of-the-art alternatives such as `wfmash+seqwish`. Replacing the latter with `AlfaPang` in the `pggb` pipeline results in output graphs with similar properties in terms of graph structure, but with a larger number of aligned genome residues. Although the decision whether or not given fragments of genomic sequences should be aligned in a pangenome graph is somewhat arbitrary, this fact reflects the high sensitivity of `AlfaPang` to sequence similarity.

Another state-of-the-art pangenome graph builder tool, `Minigraph-Cactus`, substantially differs from both `pggb` and `AlfaPang` in the assumptions on how the pangenome graph should look like. First, it does not align neither paralogs (i.e. similar sequence fragments in the same genome) nor homologuous sequences that in different genomes occur on different chromosomes. Second, it requires the user to choose a reference genome. This choice highly influences the output graph, as it may ignore similarity between fragments of non-reference genomes that have no homologs in the reference. Both design assumptions make it possible to reduce the number of sequence alignments necessary to build the graph, which consequently allows

Minigraph-Cactus to provide much better computational efficiency than pggb. However, the alignment-free approach of AlfaPang allows even higher reduction, especially in terms of required memory.

The close relationship between the variation graphs constructed by AlfaPang and the de Bruijn graphs provides a bridge between both pangenome models. On the other hand, this is a limitation of the AlfaPang approach, as the structure of resulting graphs resembles the structure of de Bruijn graphs with their drawbacks, such as excessive entanglement in areas representing low-complexity sequence regions. Such entanglement is removed in the refinement step of the AlfaPang+ pipeline by the smoothxg tool. However, due to high AlfaPang efficiency, this step dominates the whole AlfaPang+ computation time ($\sim 95\%$ on all datasets). Perhaps more precise tuning of the smoothxg parameters would allow to reduce this time without affecting the output. More substantial reduction would probably require incorporation of the refinement procedure in the graph building process of AlfaPang.

References

1 Francesco Andreace, Pierre Lechat, Yoann Dufresne, and Rayan Chikhi. Comparing methods for constructing and representing human pangenome graphs. *Genome Biology*, 24(1), 2023. doi:10.1186/s13059-023-03098-2.

2 Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.

3 Jasmijn A Baaijens, Bastiaan Van der Roest, Johannes Köster, Leen Stougie, and Alexander Schönhuth. Full-length de novo viral quasispecies assembly through variation graph construction. *Bioinformatics*, 35(24):5086–5094, December 2019. doi:10.1093/bioinformatics/btz443.

4 Adam Cicherski. AlfaPang. Software, version 1.0., This work was supported by the National Science Centre, Poland, under grant number 2022/47/B/ST6/03154, swhId: swh:1:dir:e8a27a620673d796d0701ab29a39aa2383bece22 (visited on 2024-08-16). URL: https://github.com/AdamCicherski/AlfaPang.

5 Adam Cicherski and Norbert Dojer. From de bruijn graphs to variation graphs – relationships between pangenome models. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval 2023*, pages 114–128, Cham, 2023. Springer Nature Switzerland.

6 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Brief Bioinformatics*, 19(1):118–135, January 2018. doi:10.1093/bib/bbw089.

7 Dent Earl, Ngan Nguyen, Glenn Hickey, Robert S Harris, Stephen Fitzgerald, Kathryn Beal, Igor Seledtsov, Vladimir Molodtsov, Brian J Raney, Hiram Clawson, Jaebum Kim, Carsten Kemena, Jia-Ming Chang, Ionas Erb, Alexander Poliakov, Minmei Hou, Javier Herrero, William James Kent, Victor Solovyev, Aaron E Darling, Jian Ma, Cedric Notredame, Michael Brudno, Inna Dubchak, David Haussler, and Benedict Paten. Alignathon: a competitive assessment of whole-genome alignment methods. *Genome Res*, 24(12):2077–2089, October 2014.

8 Jack Edmonds and Ellis L. Johnson. Matching: A well-solved class of integer linear programs. In Michael Jünger, Gerhard Reinelt, Giovanni Rinaldi, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, editors, *Combinatorial optimization —eureka, you shrink! papers dedicated to jack edmonds 5th international workshop aussois, france, march 5–9, 2001 revised papers*, volume 2570 of *Lecture notes in computer science*, pages 27–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/3-540-36478-1_3.

9 Jordan M Eizenga, Adam M Novak, Jonas A Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D Seaman, Robin Rounthwaite, Jana Ebler, Mikko Rautiainen, Shilpa Garg, Benedict Paten, Tobias Marschall, Jouni Sirén, and Erik Garrison. Pangenome graphs. *Annu Rev Genomics Hum Genet*, 21:139–162, August 2020. doi:10.1146/annurev-genom-120219-080406.

10 Shilpa Garg, Renzo Balboa, and Josiah Kuja. Chromosome-scale haplotype-resolved pangenomics. *Trends in Genetics*, 38(11):1103–1107, November 2022. doi:10.1016/j.tig.2022.06.011.

11 Erik Garrison and Andrea Guarracino. Unbiased pangenome graphs. *Bioinformatics*, 39(1), January 2023. doi:10.1093/bioinformatics/btac743.

12 Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, David G. Ashbrook, Kaisa Thorell, Rachel L. Rusholme-Pilcher, Gianni Liti, Emilio Rudbeck, Sven Nahnsen, Zuyu Yang, Mwaniki N. Moses, Franklin L. Nobrega, Yi Wu, Hao Chen, Joep de Ligt, Peter H. Sudmant, Nicole Soranzo, Vincenza Colonna, Robert W. Williams, and Pjotr Prins. Building pangenome graphs. *bioRxiv*, 2023. doi:10.1101/2023.04.05.535718.

13 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat Biotechnol*, 36(9):875–879, October 2018. doi:10.1038/nbt.4227.

14 Andrea Guarracino, Simon Heumos, Sven Nahnsen, Pjotr Prins, and Erik Garrison. ODGI: understanding pangenome graphs. *Bioinformatics*, 38(13):3319–3326, June 2022. doi:10.1093/bioinformatics/btac308.

15 Glenn Hickey, Jean Monlong, Jana Ebler, Adam M. Novak, Jordan M. Eizenga, Yan Gao, Haley J. Abel, Lucinda L. Antonacci-Fulton, Mobin Asri, Gunjan Baid, Carl A. Baker, Anastasiya Belyaeva, Konstantinos Billis, Guillaume Bourque, Silvia Buonaiuto, Andrew Carroll, Mark J. P. Chaisson, Pi-Chuan Chang, Xian H. Chang, Haoyu Cheng, Justin Chu, Sarah Cody, Vincenza Colonna, Daniel E. Cook, Robert M. Cook-Deegan, Omar E. Cornejo, Mark Diekhans, Daniel Doerr, Peter Ebert, Jana Ebler, Evan E. Eichler, Susan Fairley, Olivier Fedrigo, Adam L. Felsenfeld, Xiaowen Feng, Christian Fischer, Paul Flicek, Giulio Formenti, Adam Frankish, Robert S. Fulton, Shilpa Garg, Erik Garrison, Nanibaa' A. Garrison, Carlos Garcia Giron, Richard E. Green, Cristian Groza, Andrea Guarracino, Leanne Haggerty, Ira M. Hall, William T. Harvey, Marina Haukness, David Haussler, Simon Heumos, Kendra Hoekzema, Thibaut Hourlier, Kerstin Howe, Miten Jain, Erich D. Jarvis, Hanlee P. Ji, Eimear E. Kenny, Barbara A. Koenig, Alexey Kolesnikov, Jan O. Korbel, Jennifer Kordosky, Sergey Koren, HoJoon Lee, Alexandra P. Lewis, Wen-Wei Liao, Shuangjia Lu, Tsung-Yu Lu, Julian K. Lucas, Hugo Magalhães, Santiago Marco-Sola, Pierre Marijon, Charles Markello, Tobias Marschall, Fergal J. Martin, Ann McCartney, Jennifer McDaniel, Karen H. Miga, Matthew W. Mitchell, Jacquelyn Mountcastle, Katherine M. Munson, Moses Njagi Mwaniki, Maria Nattestad, Sergey Nurk, Hugh E. Olsen, Nathan D. Olson, Trevor Pesout, Adam M. Phillippy, Alice B. Popejoy, David Porubsky, Pjotr Prins, Daniela Puiu, Mikko Rautiainen, Allison A. Regier, Arang Rhie, Samuel Sacco, Ashley D. Sanders, Valerie A. Schneider, Baergen I. Schultz, Kishwar Shafin, Jonas A. Sibbesen, Jouni Sirén, Michael W. Smith, Heidi J. Sofia, Ahmad N. Abou Tayoun, Françoise Thibaud-Nissen, Chad Tomlinson, Francesca Floriana Tricomi, Flavia Villani, Mitchell R. Vollger, Justin Wagner, Brian Walenz, Ting Wang, Jonathan M. D. Wood, Aleksey V. Zimin, Justin M. Zook, Tobias Marschall, Heng Li, and Benedict Paten. Pangenome graph construction from genome alignments with minigraph-cactus. *Nature Biotechnology*, 42(4):663–673, May 2023. doi:10.1038/s41587-023-01793-w.

16 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome Biology*, 21(1), September 2020. doi:10.1186/s13059-020-02135-8.

17 Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1):i177–i186, July 2021. `doi:10.1093/bioinformatics/btab309`.

18 Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, December 2017. `doi:10.1093/bioinformatics/btw609`.

19 Samuel O'Donnell, Jia-Xing Yue, Omar Abou Saada, Nicolas Agier, Claudia Caradec, Thomas Cokelaer, Matteo De Chiara, Stéphane Delmas, Fabien Dutreux, Téo Fournier, Anne Friedrich, Etienne Kornobis, Jing Li, Zepu Miao, Lorenzo Tattini, Joseph Schacherer, Gianni Liti, and Gilles Fischer. Telomere-to-telomere assemblies of 142 strains characterize the genome structural landscape in saccharomyces cerevisiae. *Nature Genetics*, 55(8):1390–1399, August 2023. `doi:10.1038/s41588-023-01459-y`.

20 Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome Res*, 27(5):665–676, May 2017. `doi:10.1101/gr.214155.116`.

21 Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biol*, 21(1):253, September 2020. `doi:10.1186/s13059-020-02157-2`.