PanCake: A Data Structure for Pangenomes

Corinna Ernst and Sven Rahmann

Genome Informatics, Institute of Human Genetics, Faculty of Medicine
University of Duisburg-Essen, Essen, Germany
{corinna.ernst,sven.rahmann}@uni-due.de

Abstract

We present a pangenome data structure ("PanCake") for sets of related genomes, based on bundling similar sequence regions into shared features, which are derived from genome-wide pairwise sequence alignments. We discuss the design of the data structure, basic operations on it and methods to predict core genomes and singleton regions. In contrast to many other pangenome analysis tools, like EDGAR or PGAT, PanCake is independent of gene annotations. Nevertheless, comparison of identified core and singleton regions shows good agreements. The PanCake data structure requires significantly less space than the sum of individual sequence files.

1998 ACM Subject Classification E.2 Data Storage Representations, J.3 Life and Medical Sciences

Keywords and phrases pangenome, data structure, core genome, comparative genomics

Digital Object Identifier 10.4230/OASIcs.GCB.2013.35

1 Introduction

With an increasing amount of available sequence data, biological research shifts towards the exploration of the global gene repertoire of related species, called the pangenome, instead of single genomic sequences. The term "pangenome" was introduced in 2005 in a study that compared eight strains of *Streptococcus agalactiae* [13], identified roughly 1800 genes shared by all strains and defined them as the "core genome". The core genome is assumed to consist mainly of genes regulating essential life processes, and hence being indispensable to cell survival. Genes only present in a subset of genomes were called "dispensable" [13]. Analysis of the pangenome of a set of related prokaryotic strains yields insight in the delineation of species and can be taken as a basis for taxonomic classification [10]. Genes identified as specific to a single genome, called "singletons", act as candidates accountable for strain-specific characteristics like virulence or synthesis of certain metabolites [9, 14].

Several tools for the analysis of pangenomes exist. Many of them require pre-computed information stored in databases [2, 4, 5]. Consequently these tools are applicable only for a set of provided strains. Furthermore, most applications rely on the availability of gene annotations [2, 15], which may not be on hand in all cases, or incomplete, or erroneous [11, 12].

The pangenome concept can be extended to the level of pure genomic sequences without annotations. Information about pairwise local sequence similarities can be obtained from alignment tools like `BLAST` [1] or `nucmer` [7]. Based on pairwise similarities, Mancheron et al. [9] introduced an approach for the identification of regions shared by all input sequences, which was subsequently improved by Jahn et al. [6]. Regions that appear similar in all compared sequences are expected to be part of the core genome, while putative singletons lie in areas not aligned to any of the other genomes. Core and singleton identification based on pairwise alignments is independent from annotaions, is able to handle gene duplication events and can even serve as a resource for annotation refinement [9].

We present a novel approach for the analysis of pangenomes based on pooling related genomic subsequences into common objects, which we call *shared features*. Pangenome-related information of the genomes (such as core regions) can be derived directly from shared features. Additionally, storage requirements are reduced in comparison to pure sequences, even without using explicit compression, but by storing similar sequences via sequences of edit operations with respect to a common reference [3, 8].

In Section 2 we introduce the PanCake data structure, especially shared features and feature instances. In Section 3 we explain our approach of decoding sequences as edit operations with respect to a given reference. In Section 4 we show how the data structure is built iteratively from pairwise alignments between the included genome sequences. In Section 5 we explain how the data structure is used to identify core and singleton regions. Section 6 briefly describes the PanCake software. In Section 7 we report results on strains from three different prokaryotic genera by comparing our findings with those of pangenome analysis tools PGAT [4] and EDGAR [2]. A discussion with outlook concludes the paper.

2 The PanCake Data Structure

In our model, a *pangenome* P consists of $n_\mathrm{g} \geq 1$ genomes. Each genome consists of one or several chromosomes, such that the pangenome consists in total of $n_\mathrm{c} \geq n_\mathrm{g}$ chromosomes. The sequence of chromosome C between positions p and q, inclusive, is written as $C[p:q]$.

The centerpiece of our approach is the bundling of similar subsequences from diverse genomes into a common object, which we call a *shared feature*.

▶ **Definition 1** (pangenome)**.** A *pangenome* consists of a set of genomes, a set of chromosomes, a mapping of chromosomes to genomes, and a set of shared features (Definition 2).

Each shared feature consists of one reference sequence r and a non-empty set of so-called *feature instances*. An example of a shared feature with four feature instances is shown in Figure 1 (left). Each feature instance represents a single genomic interval on a chromosome.

▶ **Definition 2** (shared feature)**.** A *shared feature* is a pair $(r, \mathcal{F})$, consisting of a DNA reference sequence r and a set $\mathcal{F}$ of feature instances (Definition 3).

▶ **Definition 3** (feature instance)**.** A *feature instance* $F = (C, \mathit{start}, \mathit{stop}, S, e, b, \mathit{prev}, \mathit{next})$ consists of a chromosome identifier C with start and stop positions $\mathit{start} \leq \mathit{stop}$ on C. Further, S references the shared feature F is organized in, e is the sequence of edit operations which have to be applied to S's reference sequence r to obtain the feature instance's sequence, and b is a *direction bit* that takes the values *forward* or *reverse*. If the direction bit is *forward*, application of e to the reference sequence results in the chromosome sequence $C[\mathit{start}: \mathit{stop}]$ directly, otherwise in its reverse complement. The feature instances belonging to the same chromosome are organized as a doubly linked list, with *prev* pointing to the previous (upstream) feature instance ending at position $\mathit{start} - 1$, and *next* pointing to the next (downstream) one starting at position $\mathit{stop} + 1$. (At the chromosome telomers, these take a special null value.)

Any chromosome C can be reconstructed by iterating over linked feature instances, starting from the feature instance covering the chromosome's start and stopping at its end. Linearly iterating over a linked list to access a specific chromosome position can be slow, so we use an index that maps each Δ-th position of a chromosome to the feature instance covering it, for navigation within the data structure. Currently, we use $\Delta = 10\,000$.

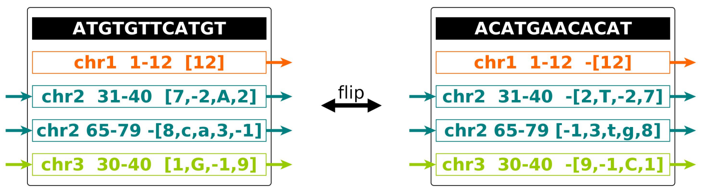

Figure 1 A shared feature before and after reverse-complementing (flipping) with reference sequence `ATGTGTTCATGT` or rev. complement `ACATGAACACAT` and four feature instances (Definition 3). The direction bit is shown as a minus sign in front of the list of edit operations if it is *reverse*. Colored arrows represent links to next and from previous feature instances. The feature instance covering chr1 is not linked to an upstream feature instance because it covers the chromosome's start.

If a shared feature contains only a single feature instance (as all do initially, when no sequence similarities have been detected and processed), we dispense with the overhead of shared features and store such an instance as an *unaligned feature instance*, which is a tuple $(C, start, stop, s, prev, next)$, whereby definitions of C, $start$, $stop$, $prev$ and $next$ are the same as in Definition 3 and s directly spells the covered $C[start : stop]$ (instead of representing it indirectly via a reference r of a shared feature S and edit operations e and a direction bit b).

3 Sequence Encoding by Edit Operations

As explained in Definition 3, the chromosomal sequence s of a feature instance is encoded through a reference sequence r (of the enclosing shared feature) and edit operations e plus a direction bit b. In Section 3.1 we explain how e is derived from a pairwise alignment of r and s. However, if (e.g. for incorporation of similarity information) a feature instance has to be moved from one shared feature to another, its edit operations have to be adapted (rebased) to a new reference sequence without explicit knowledge of the corresponding pairwise alignment. Our rebasing approach is discussed in Section 3.2.

3.1 Deriving Edit Operations from Pairwise Alignments

Edit operations describe the alignment of a feature instance's sequence to the reference sequence of its enclosing shared feature by using the standard operations (match, substitution, insertion, deletion) on single characters. They can be encoded efficiently as byte sequences, as each represented DNA sequence is assumed to be closely similar to the reference. Therefore one can store bytes with the most significant bit deciding wheter a number or character is stored. If a number is stored, the second most significant bit determines the sign. A positive number indicates consecutive matches; a negative number indicates consecutive deletions. If a character is stored, we store its ASCII code in seven bits: Substitutions are encoded by the uppercase IUPAC symbol of the substituted nucleic acid, insertions are encoded by lower-case symbols. In this paper, we show edit operations as lists of numbers and IUPAC symbols, prepending them by a minus sign if and only if the direction bit is *reverse*. It is straightforward to convert between a sequence of edit operations and an alignment, as edit operations are simply a compact encoding of the alignment, given the original sequence. We summarize both e and the direction bit b as an edit transformation $t = (e, b)$ and write $r \overset{t}{\mapsto} s$. Such a transformation is equivalent to a pairwise alignment of r and s, or of r and the reverse complement of s.

```
r' AT-GTGTTCATGT        A-T-GTGTTCATGT
r  ATAGTGTT-A-GT        A-TAGTGTT-A-GT
                   ⇒                       ⇒   A-T-GTGTTCATGT   r'
r  A-TAGTGTTAGT         A-TAGTGTT-A-GT         AGTCGT-TT-A-GT   s
s  AGTCGT-TTAGT         AGTCGT-TT-A-GT
```

$$\begin{tikzcd} r' \arrow[r, "t^*"] \arrow[d, "e'"] & r'' \arrow[d, "e''"'] \\ s' \arrow[r, "a"] & s'' \end{tikzcd}$$

Figure 2 Left: Rebasing s on a new reference r' from previous reference r when a transformation (alignment) between r' and r is known (left): Gaps in both alignments are expanded to their union, such that the gapped representation of the old reference r becomes equal in both alignments (middle) This directly results in an alignment between r' and s by forgetting r (right). Right: Commutative diagram showing how to find a transformation t^* between two reference sequences $r^* = r'$ and r'', when transformations $r' \overset{e'}{\mapsto} s'$, $s' \overset{a}{\mapsto} s''$ and $r'' \overset{e''}{\mapsto} s''$ are given: $t^* = e' \cdot a \cdot e''^{-1}$.

3.2 Rebasing Edit Operations on a Different Reference Sequence

If we have to rebase a feature instance's sequence s on a different reference sequence r', we need to find a transformation t' such that $r' \overset{t'}{\mapsto} s$. Of course, we could compute an optimal alignment between r' and s (or its reverse complement) from scratch. However, this would be time-consuming, and we can assume that we already have an alignment (or transformation u) between r' and original reference r. Algebraically, if we have $r' \overset{u}{\mapsto} r$ and $r \overset{t}{\mapsto} s$, we can compose them to $r' \overset{u}{\mapsto} r \overset{t}{\mapsto} s$, and we will write $t' = u \cdot t$. We will also use the multiplicative notation if a transformation is applied to a sequence, i.e. $r' \cdot t' = r' \cdot u \cdot t = s$. This notation is completely analogous that of linear algebra, multiplying a row vector r' with several (size-compatible) matrices u, t, obtaining a new row vector s. Of course, the operations have nothing in common with matrix multiplication; we simply borrow the notational convenience.

The transformation t' corresponds to a pairwise alignment between r' and s, but not necessarily an optimal one, even if transformations t and u are optimal. The process of composing edit transformations can be understood in terms of pairwise alignements, as explained in Figure 2. In some cases, the resulting alignment or transformation can be simplified directly: In the alignment, columns of gap aligned to gap are removed. Insertions directly followed by deletions (or vice versa) can be converted to substitutions.

4 Building the PanCake Data Structure

Initially, each of the n_c chromosomes in a PanCake data structure is represented by its own shared feature and feature instance. During an iterative building process, feature instances are bundled into shared features on the basis of pairwise alignments computed by external tools. Section 4.1 describes how similarity information arising from a single pairwise alignment is integrated into the data structure. In summary, integration occurs in three steps, namely division (Section 4.2), (conditional) flipping (Section 4.3), and merging (Section 4.4) of shared features.

4.1 Including a Pairwise Alignment into the PanCake Data Structure

Independently of using `BLAST` [1] or `nucmer` [7] for computation, PanCake represents a pairwise alignment as follows.

▶ **Definition 4** (PanCake pairwise alignment)**.** PanCake describes a pairwise alignment $A = (A_1, A_2)$ between two chromosomal intervals by two 5-tuples $A_i = (C_i, start_i, stop_i, b_i, s_i)$ with $i \in \{1, 2\}$. Here $C_i[start_i : stop_i]$ defines the i-th sequence by specifying its chromosome,

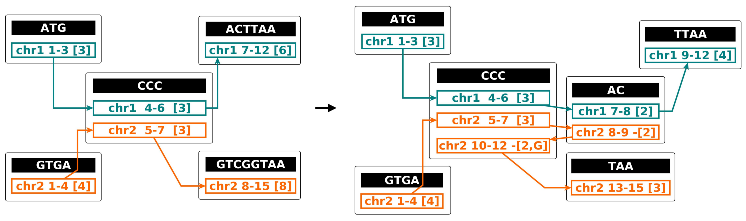

Figure 3 Inclusion of pairwise alignment A into an initial PanCake data structure containing two artificial chromosmomes of length 12bp and 15bp and consisiting of six feature instances organized in five shared features. Let $A = ((\text{chr1}, 4, 8, \mathit{forward}, \texttt{CCCAC}), (\text{chr2}, 8, 12, \mathit{reverse}, \texttt{CCGAC}))$. As the aligned subsequence chr1[4:8] spans more than one feature instance initially, A is divided into $A' = ((\text{chr1}, 4, 6, \mathit{forward}, \texttt{CCC}), (\text{chr2}, 10, 12, \mathit{reverse}, \texttt{CCG}))$ and $A'' = ((\text{chr1}, 7, 8, \mathit{forward}, \texttt{AC}), (\text{chr2}, 8, 9, \mathit{reverse}, \texttt{AC}))$. Then, A' and A'' are integrated separately.

start and stop position ($C_1 = C_2$ is possible). If b_i is *forward*, that sequence is aligned, otherwise its reverse complement. The rows of the alignment (sequences with gap characters) are given by s_1, s_2.

To incorporate the information of a pairwise alignment $A = ((C_1, \mathit{start}_1, \mathit{stop}_1, b_1, s_1), (C_2, \mathit{start}_2, \mathit{stop}_2, b_2, s_2))$ into the data structure, we proceed as follows. We find the (at most four different) feature instances F_1^{start}, F_1^{stop}, F_2^{start} and F_2^{stop}, covering positions $C_1[\mathit{start}_1]$, $C_1[\mathit{stop}_1]$, $C_2[\mathit{start}_2]$ and $C_2[\mathit{stop}_2]$, respectively. We divide F_1^{start} at $C_1[\mathit{start}_1]$ according to Section 4.2 (unless the feature instance ends at that position anyway) and do so analogously for the other feature instances and corresponding positions.

If, thereafter, $C_1[\mathit{start}_1 : \mathit{stop}_1]$ or $C_2[\mathit{start}_2 : \mathit{stop}_2]$ spans more than a single feature instance, we divide the alignment A into several disjoint alignments such that for each resulting subalignment $A' = ((C_1', \mathit{start}_1', \mathit{stop}_1', b_1', s_1'), (C_2', \mathit{start}_2', \mathit{stop}_2', b_2', s_2'))$, chromosmal region $C_1'[\mathit{start}_1' : \mathit{stop}_1']$ is covered entirely by a feature instance F_1 and chromosomal region $C_2'[\mathit{start}_2' : \mathit{stop}_2']$ is covered entirely by F_2. We then merge the shared features containing F_1 and F_2 into a single one according to Section 4.4. Depending on the direction bits of the alignment, one shared feature may have to be flipped before (Section 4.3). If division results in a subalignment A' with either s_1' or s_2' consisting exclusively of gaps, then this subalignment is discarded, and no merge is performed. An example is shown in Figure 3.

The resulting data structure depends on the order in which the alignments are processed.

4.2 Dividing a Shared Feature and its Feature Instances

Dividing a feature instance into two disjoint parts implies the division of its containing shared feature and hence all other contained feature instances, too.

Given a feature instance $F = (C, \mathit{start}, \mathit{stop}, S, e, b, \mathit{prev}, \mathit{next})$ and a cutting index c with $1 \le c \le \mathit{stop} - \mathit{start} + 1$, the task is to divide F at distance c from the start or stop position, depending on the direction bit b. This results in F being divided into two new feature instances F' and F'' and corresponding new shared features S' and S''.

Precisely, if b is *forward*, this results in two new feature instances

$$\begin{aligned} F' &= (C, \quad \mathit{start}, \quad \mathit{start} + c - 1, \quad S', \quad e', \quad \mathit{forward}, \quad \mathit{prev}, \quad F''), \\ F'' &= (C, \quad \mathit{start} + c, \quad \mathit{stop}, \quad S'', \quad e'', \quad \mathit{forward}, \quad F', \quad \mathit{next}). \end{aligned}$$

Otherwise, if b is *reverse*, this results in

$$\begin{aligned} F' &= (C, \quad \mathit{start}, \quad \mathit{stop} - c, \quad S', \quad e', \quad \mathit{reverse}, \quad \mathit{prev}, \quad F''), \\ F'' &= (C, \quad \mathit{stop} - c + 1, \quad \mathit{stop}, \quad S'', \quad e'', \quad \mathit{reverse}, \quad F', \quad \mathit{next}). \end{aligned}$$

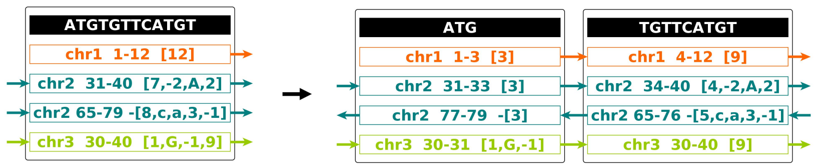

Figure 4 Division of the orange feature instance $F = (\text{chr1}, 1, 12, S, [12], \textit{forward}, \textit{prev}, \textit{next})$ at cutting index $c = 3$. Computing the cutting indexes for the reference r and all other feature instances $\tilde{F} \in S$ results in $c_r = 3$ and $c_{\tilde{F}} = 3$ except instance $(\text{chr3}, 30, 40, \dots)$, where $c_{\tilde{F}} = 2$. The newly formed shared features are *concatenated shared features*, cf. Section 5.

Note that *all* feature instances of the corresponding shared feature S have to be divided as well to maintain the data structure. An example is given in Figure 4.

To divide the containing shared feature S, we must compute the position c_r at which the reference r of S has to be divided, such that the references of S', S'' are $r' = r[1 : c_r]$ and $r'' = r[(c_r + 1) : |r|]$. Computation of c_r from c and implicit sequence s of F proceeds by counting positions in the implicit aligment represented by edit transformation $r \overset{(e,b)}{\mapsto} s$ until the length of the processed part of s becomes $\geq$ c.

Once c_r is known, divided shared features S', S'' are initialized with the prefix and suffix of the reference sequence, respectively. Their feature instances are included successively while iterating over all feature instances in S. For each feature instance $\tilde{F} \in \mathcal{F} \setminus F$, splitting position $\tilde{c}$ is determined (analogously to computation of c_r) in order to compute new edit operation lists $\tilde{e}', \tilde{e}''$ and adapt the start and stop positions of the newly formed feature instances. Start and stop positions of the divided feature instances $\tilde{F}'$ and $\tilde{F}''$, as well as their links to next and previous feature instances, depend on $\tilde{F}$'s direction bit $\tilde{b}$.

During division, two special cases may arise. First, empty feature instances $\tilde{F}'$ or $\tilde{F}''$ with edit operations consisting of only deletions may occur. Such feature instances are deleted entirely and links from previous and next instances adjusted accordingly. Second, an empty reference sequence $r_{S'}$ or $r_{S''}$ may occur if the beginning or end of F's edit operation list e consists exclusively of insertions. Then any feature instance is chosen (e.g., randomly) whose decoded sequence provides the new reference. The edit operations of the remaining feature instances are then rebased on the new reference (Section 3.2).

Dividing a shared feature and its feature instances requires updating the navigation index if for any chromosome, a position divisible by Δ belonged to the divided S.

4.3 Flipping a Shared Feature

Flipping a shared feature $S = (r, \mathcal{F})$ works as follows. After reverse-complementing the reference sequence r, new edit operations are computed for each feature instance $F \in \mathcal{F}$ by reversing the sequence, reverse-complementing symbols referring to substitutions or insertions, and finally flipping F's direction bit. By flipping, none of the chromosome sequences change, but only their representation by a reference sequence and edit operations. Applying flipping twice results in the original representation.

4.4 Merging Shared Features

When two shared features $S' = (r', \mathcal{F}')$ and $S'' = (r'', \mathcal{F}'')$ have similar reference sequences $r' \approx r''$ (e.g., as evidenced by finding a good alignment between feature instances of $S' \neq S''$), it is beneficial to merge S' and S'' into a new combined shared feature $S^* = (r^*, \mathcal{F}' \cup \mathcal{F}'')$

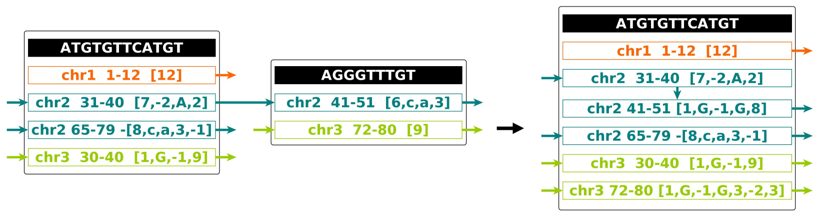

Figure 5 Merging two shared features, assuming that an alignment between chr2, positions 41–51, and chr3, positions 30–40, has been found.

containing all feature instances from both S' and S''. Assume without loss of generality that S' is larger, i.e., $|\mathcal{F}'| \geq |\mathcal{F}''|$. Then we pick $r^* := r'$ as reference and move each $F' \in \mathcal{F}'$ into $\mathcal{F}^*$ unmodified. We have to base each feature instance of S'' on $r^* = r'$. This works as explained in Section 3.2, but we need to know a transformation t^* such that $r' \overset{t^*}{\mapsto} r''$. In a typical case, we do not have such a transformation available directly, but first need to compute it from a given alignment (e.g., found by BLAST) between two feature instances F' and F''. Say the rows of the pairwise alignment are given by s' and s'', corresponding to an edit transformation a. We also know edit transformations $r' \overset{e'}{\mapsto} s'$ and $r'' \overset{e''}{\mapsto} s''$ stored in the feature instances. Now $t^* = e' \cdot a \cdot e''^{-1}$, as shown by the commutative diagram in Figure 2 (Right). Here e''^{-1} denotes the inverse edit transformation, replacing insertions by deletions and vice versa and swapping goal and target of substitutions. An example of the result of merging two shared features is provided in Figure 5.

Merging two shared features requires updating the navigation index for those positions divisible by Δ contained in the smaller S''.

5 Applications: Core and Singleton Identification

Once a PanCake representation of several genomes is built and stored, there are several applications; the two most important of which are the identification of singletons (here meaning genomic regions not shared with any other genome) and of the core genome (here referring to genomic regions shared with every other genome).

Singleton identification is straightforward. By definition, the set of singleton regions of a genome G consists of all *unaligned* feature instances belonging to chromosomes of G (see Section 2) or being part of a shared feature containing exclusively feature instances originating from G. They can be easily enumerated by iterating over all feature instances of chromosomes of G.

The core genome can be identified by considering core features, defined as follows.

▶ **Definition 5** (core feature). A $\mathcal{G}$-*core feature* for a set $\mathcal{G}$ of genomes is a shared feature that contains at least one feature instance from each genome $G \in \mathcal{G}$.

Identification and enumeration of core features is straightforward by using the map of contained chromosomes to genomes. If we only need to know which positions in a genome belong to the core, we are done now. However, we additionally want to list all (maximal) core regions whose definition corresponds to maximum common intervals (MCIs) by Mancheron et al. [9] or maximum overlapping intervals (MOIs) by Jahn et al. [6]. We first need the notion of concatenated shared features.

▶ **Definition 6** (concatenated shared features). An ordered pair (S', S'') of shared features is *concatenated*, if they have the same number of feature instances and for all feature instances

F' in S' with direction bit *forward*, there exists a feature instance F'' in S'' with $next' = F''$ and direction bit *forward*, and for all F' in S' with direction bit *reverse*, there exists F'' in S'' with $next'' = F'$ and direction bit *reverse*. (S', S'') is also called concatenated if any combination of flipping results in concatenation.

Concatenated shared features can arise from dividing a shared feature (Section 4.2; Figure 4). The concept is iteratively extended to more than two shared features. We now define what it means that a sequence of consecutive feature instances from the same chromosome forms a *core region part* and then define a *core region* as a maximal core region part.

▶ **Definition 7** (core region part)**.** Let $\mathcal{G}$ be a set of genomes. Let F_i, $1 \le i \le n$ be consecutive feature instances (meaning that $stop_i + 1 = start_{i+1}$ for all i) on a chromosome C from genome $G \in \mathcal{G}$. Let $S_i = (r_i, \mathcal{F}_i)$ be the shared feature containing F_i.

The F_i, $1 \le i \le n$ (and the induced interval $C[start_1, stop_n]$) is a *core region part of* G with respect to $\mathcal{G}$, if for all $1 \le i \le n$, there exists a subset of feature instances, $\hat{\mathcal{F}}_i \subseteq \mathcal{F}_i$, such that: (a) the reduced shared features $\hat{S}_i := (r_i, \hat{\mathcal{F}}_i)$ are concatenated shared features, and (b) for each genome $G' \in \mathcal{G}$, there exists at least one feature instance in $\hat{\mathcal{F}}_i$ covering a chromosome from G'.

Every core feature gives rise to a core region part for each of its contained feature instances, but a core region part can be longer than a single feature instance. Our interest lies in connecting these parts to a (full) core region which cannot be extended further.

▶ **Definition 8** (core region)**.** A core region part (F_i), $1 \le i \le n$, of C or G is a *core region* of C or G if it is maximal in the sense that both upstream elongation (by prepending $prev_1$) and downstream elongation (by appending $next_n$) do not yield core region parts.

Identification of core regions of a chromosome C with respect to a genome set $\mathcal{G}$ occurs in two steps. First, starting from each C-covering feature instance F contained in a core feature with respect to $\mathcal{G}$, we search for the longest core region part of C starting with F. In a second step, whenever identified core region parts share stop positions on the reference genome, shorter ones are removed from output.

In practice, there exist short unaligned feature instances (e.g., 1–5 bp) between otherwise concatendated shared features, breaking the concatenation property. As this results in unnecessary cutting of long core regions, we have implemented a gap-tolerant version of the above method that ignores intermediate feature instances shorter than a user-defined length.

6 The PanCake Software

The PanCake software is written in Python 3.2 and available from `https://bitbucket.org/CorinnaErnst/pancake` under the MIT license. It has a command-line interface with several subcommands, allowing to add chromosomes from `.fasta` files, to specify a genome for each chromosome, to add alignments, to compute core and singleton regions, and to output selected subsequences of the contained chromosomes. Intermediate representations of the data structure are serialized into a text-based file format (PanCake `.pan` format), which is manipulated by these subcommands. Even though unoptimized, significant savings against the `.fasta` files are visible (cf. Table 1). Core genome computation outputs a `.bed` file with intervals for each chromosome covering the core regions, and optionally one `.fasta` file per core region, containing their unaligned sequences, so they can be optimally aligned and inspected with standard tools.

Table 1 PanCake statistics on three genera (see text). Computation time refers to a single core on an Intel Core i7-2600 CPU at 3.40GHz with 8 GB RAM.

Genus (number of strains)	genome size [Mbp]	FASTA size [MB]	number of alignments	PanCake size [MB]	comp time
Pseudomonas (3)	19.4	19.7	1405	7.7 (39%)	20 sec
Yersinia (8)	38.0	38.5	324925	8.9 (23%)	16.5 h
Burkholderia (10)	65.3	66.2	147344	22.0 (33%)	10.8 h

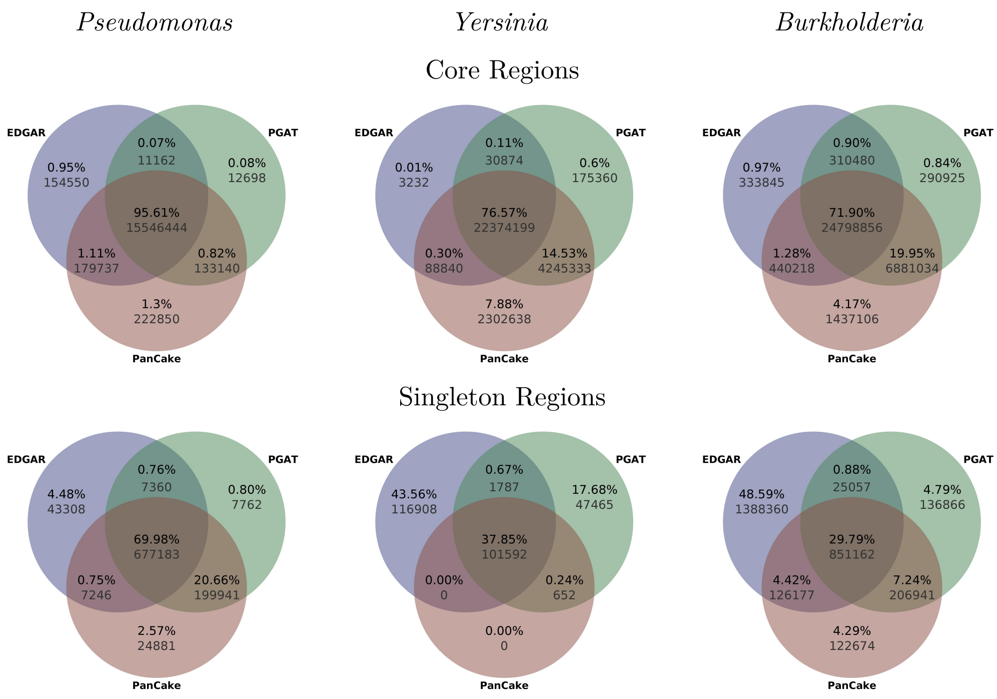

Figure 6 Overlap of core and singleton regions as identified by EDGAR, PGAT and PanCake on the datasets of Table 1.

7 Results

We compare the results of PanCake against those of two other comparative genome analysis tools: EDGAR [2] and PGAT [4]. Both approaches identify core genes and singletons by comparing pre-annotated coding sequences only, while PanCake does whole-genome comparisons. As EDGAR and PGAT are web-based database applications, analysis is limited to the sets of provided pre-processed strains, at least in open access mode. PGAT provides 8 bacterial genera, from which we chose *Pseudomonas*, *Yersinia* and *Burkholderia*; we exclude strains marked as draft assembly and strains with chromosomes or plasmids which are absent in EDGAR's open access mode. This results in the following sets of strains:

Pseudomonas (3 strains): *P. aeruginosa PAO1*, *P. aer. UCBPP-PA14*, *P. aer. LESB58*.

Yersinia (8 strains): *Y. pestis Angola*, *Y. pestis Antiqua*, *Y. pestis KIM 10*, *Y. pestis Microtus 91001*, *Y. pestis Nepal 516*, *Y. pestis Pestoides F*, *Y. pestis Z176003*, *Y. pestis CO92*.

Burkholderia (10 strains): *B. pseudomallei 1026b*, *B. pseudomallei 1106a*, *B. pseudomallei 1710b*, *B. pseudomallei 668*, *B. pseudomallei K96243*, *B. mallei ATCC 23344*, *B. mallei NCTC 10229*, *B. mallei NCTC 10247*, *B. mallei SAVP1*, *B. thailandensis E264*.

With PanCake, we build a data structure for each genus, using all pairwise alignments computed by `nucmer` [7] with option `--maxmatch` (use all anchor matches regardless of their uniqueness), removing obivously redundant alignments. Some statistics are given in Table 1.

For each genus we compute the core genome and singletons, according to each tool's definitions and default parameters and recommendations, using the EDGAR web applications and PGAT's 'Prescence and Abscence Tool' with option 'consider pseudogenes as present'. As EDGAR acts exclusively on annotated genes, we only evaluate on such positions.

The results (Figure 6) show that core regions identified by the three tools are in good agreement. For *Yersinia* and *Burkholderia*, the amounts of identical core positions identified by PanCake and PGAT are higher than in combinations including EDGAR. This may be explained by EDGAR's approach of determining orthologs stringently by bidirectional best BLAST hits [2], resulting in fewer predicted core regions and more predicted singleton regions. In contrast, PGAT allows genes to be related to various similar regions in other genomes; so PGAT's results show significantly better agreement with the results of PanCake than with EDGAR. Concerning the singleton regions, only small amounts from 4.29% down to 0% of the genomic positions identified by PanCake do not agree with one of the other tools.

8 Discussion and Conclusion

We presented a data structure and software implementation (PanCake) for pangenomes. It is based on pooling similar genomic subsequences, as evidenced by pairwise alignments, into shared features. We discussed basic operations on the data structure (flipping, re-basing, division, merging) and how to iteratively build it from alignments. We also presented a method to identify the core genome and singletons from the data structure. Comparison with PGAT and EDGAR shows good agreement with PGAT, while EDGAR uses a more stringent approach to identify orthologs (instead of "similar regions"). PanCake is not restricted to annotated genes, and the data structure can be built iteratively from available (un-annotated) FASTA files and stored persistently.

In the future, we aim to reduce computation times (the current version uses un-optimized pure Python code) and storage requirements by optionally using a more efficient binary format to store the data structure. Already, the PanCake file size is only 40% to 25% of the sum of FASTA file sizes. At the moment, the resulting representation of the data structure depends on the order in which the alignments are processed (and on the quality of the alignments themselves). We are working on a refactoring operation that will provide a better representation of each shared feature (say, using a median reference sequence with short edit operation lists). New classes of shared features, such as one representing variable-length repeats, are also of interest, as well as avoiding occasional artifacts of short feature instances that result from division when an alignment does not end at, but close to, the border of a shared feature. To determine core regions faster, efficient algorithms [6, 9] could be implemented.

The current state of PanCake is a proof of concept. We plan to substantially broaden PanCake's applications by including additional features, such as better support for other alignment tools, optional inclusion of annotation data, taxonomic analysis and creation of synteny plots. A typical future query might be: Output all regions (in any genome) that are similar to the *metH* gene in any *Y. pestis* strain.

Acknowledgements We thank Johannes Köster and Marcel Martin for helpful discussions and sharing their Python knowledge.

References

1 S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

2 J. Blom, S. P. Albaum, D. Doppmeier, A. Pühler, F.-J. Vorhölter, M. Zakrzewski, and A. Goesmann. EDGAR: a software framework for the comparative analysis of prokaryotic genomes. *BMC Bioinformatics*, 10:154, 2009.

3 M. C. Brandon, D. C. Wallace, and P. Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, 2009.

4 M. J. Brittnacher, C. Fong, H. S. Hayden, et al. PGAT: a multistrain analysis resource for microbial genomes. *Bioinformatics*, 27(17):2429–2430, 2011.

5 T. Davidsen, E. Beck, A. Ganapathy, R. Montgomery, N. Zafar, Q. Yang, R. Madupu, P. Goetz, K. Galinsky, O. White, and G. Sutton. The comprehensive microbial resource. *Nucleic Acids Research*, 38(Database issue):D340–345, 2010.

6 K. Jahn, H. Sudek, and J. Stoye. Multiple genome comparison based on overlap regions of pairwise local alignments. *BMC Bioinformatics*, 13(Suppl 19):S7, 2012.

7 S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

8 P.-R. Loh, M. Baym, and B. Berger. Compressive genomics. *Nature Biotechnology*, 30(7):627–630, 2012.

9 A. Mancheron, R. Uricaru, and E. Rivals. An alternative approach to multiple genome comparison. *Nucleic Acids Research*, 39(15):e101, 2011.

10 D. Medini, C. Donati, H. Tettelin, V. Masignani, and R. Rappuoli. The microbial pan-genome. *Current Opinion in Genetics & Development*, 15(6):589–594, 2005.

11 M. S. Poptsova and J. P. Gogarten. Using comparative genome analysis to identify problems in annotated microbial genomes. *Microbiology*, 156(7):1909–1917, 2010.

12 A. M. Schnoes, S. D. Brown, I. Dodevski, and P. C. Babbitt. Annotation error in public databases: Misannotation of molecular function in enzyme superfamilies. *PLoS Computational Biology*, 5(12), 2009.

13 H. Tettelin, V. Masignani, M. J. Cieslewicz, et al. Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: implications for the microbial "pan-genome". *Proc. Natl. Acad. Sci.*, 102(39):13950–13955, 2005.

14 E. Trost, J. Blom, S. C. Soares, et al. Pangenomic study of corynebacterium diphtheriae that provides insights into the genomic diversity of pathogenic isolates from cases of classical diphtheria, endocarditis, and pneumonia. *Journal of Bacteriology*, 194(12):3199–3215, 2012.

15 Y. Zhao, J. Wu, J. Yang, S. Sun, J. Xiao, and J. Yu. PGAP: pan-genomes analysis pipeline. *Bioinformatics*, 28(3):416–418, 2012.