PanTools: representation, storage and exploration of pan-genomic data

Siavash Sheikhizadeh¹, M. Eric Schranz², Mehmet Akdel¹, Dick de Ridder¹ and Sandra Smit¹,*

¹Bioinformatics Group, Wageningen University, Droevendaalsesteeg 1, 6708PB, Wageningen, The Netherlands and ²Biosystematics Group, Wageningen University, Droevendaalsesteeg 1, 6708PB, The Netherlands

*To whom correspondence should be addressed.

Abstract

Motivation: Next-generation sequencing technology is generating a wealth of highly similar genome sequences for many species, paving the way for a transition from single-genome to pan-genome analyses. Accordingly, genomics research is going to switch from reference-centric to pan-genomic approaches. We define the pan-genome as a comprehensive representation of multiple annotated genomes, facilitating analyses on the similarity and divergence of the constituent genomes at the nucleotide, gene and genome structure level. Current pan-genomic approaches do not thoroughly address scalability, functionality and usability.

Results: We introduce a generalized De Bruijn graph as a pan-genome representation, as well as an online algorithm to construct it. This representation is stored in a Neo4j graph database, which makes our approach scalable to large eukaryotic genomes. Besides the construction algorithm, our software package, called PanTools, currently provides functionality for annotating pan-genomes, adding sequences, grouping genes, retrieving gene sequences or genomic regions, reconstructing genomes and comparing and querying pan-genomes. We demonstrate the performance of the tool using datasets of 62 E. coli genomes, 93 yeast genomes and 19 Arabidopsis thaliana genomes.

Availability and Implementation: The Java implementation of PanTools is publicly available at http://www.bif.wur.nl.

Contact: sandra.smit@wur.nl

1 Introduction

Since the assembly of the first bacterial genome in 1995 (Fleischmann et al., 1995; Fraser et al., 1995), the concept of a reference genome has been the cornerstone for gene discovery, functional analysis and comparative genomics. Reference genomes are typically linear sequence representations that facilitate genome browsing and sequence-based analyses. In recent years, large genome projects, such as the 150 tomato genome project (Aflitos et al., 2014) and the 3000 rice genomes project (Li et al., 2014), have led to a deluge of data. Thus, many species and phylogenetic groups are no longer represented by a single reference genome but by numerous related genomes. In this situation, analyzing hundreds of genomes by individual comparison to a single reference genome becomes inefficient and misses genomic content not present in the reference, while ignoring the availability of the other near-complete genomes. Likewise, pairwise comparison of hundreds of linear genomes is also far from practical. Hence, to capitalize on the genomic diversity in large collections of genomes, we need to transition from a reference-centric approach to a pan-genome approach.

Originally, the term pan-genome has been used to describe the totality of genes found in a species or phylogenetic clade in order to classify specific genes as either core or dispensable (Tettelin et al., 2005). More recent conceptions of the pan-genome are defined at the sequence level, compressing multiple genomes into a (compressed) De Bruijn graph (DBG) using additional data structures such as a suffix tree (Marcus et al., 2014), FM-index (Beller and Ohlebusch, 2015), Burrows–Wheeler transform (Baier et al., 2015) or Bloom filter trie (Holley et al., 2016). Accordingly, here we define the pan-genome as a comprehensive representation of multiple annotated genomes, facilitating analyses on the content and organization of the constituent genomes.

Aiming to replace linear genome representations, a computational pan-genome solution should contain annotations, be mutable (to incorporate novel genomes), allow long-term storage and be usable for comparative genomics. The storage property is especially important when working on large eukaryotic genomes, where in-memory solutions are no longer sufficient. Being able to update a pan-genome is essential as the rate at which genomes are produced...
will only increase in the future. Existing pan-genome approaches fulfill only part of these requirements.

To overcome these limitations and work towards a computational pan-genome approach that allows to incorporate these desirable features, we developed an algorithm to condense multiple annotated genome sequences into a single representation. As in other approaches, the core of our pan-genome is a compressed De Bruijn graph. What differentiates our method is that we construct the pan-genome in a Neo4j graph database (Have and Jensen, 2013; Robinson et al., 2013), which scales to arbitrary graph sizes and thus allows for the analysis of large collections of complex eukaryotic genomes. Our pan-genome graph is created using an online algorithm that, like current methods, has a runtime linear in the total sequence length. Besides construction, we provide useful functionality for annotating pan-genomes, grouping genes, retrieving sequences and comparing pan-genomes. The pan-genome is stored on disk and new genomes or annotated features can be added. We have implemented a stand-alone command-line Java application, called PanTools, for the representation, storage and exploration of pan-genomic data. The software is publicly available on http://www.bif.wur.nl.

This article details our algorithm for pan-genome construction and discusses its functionality, performance and applications. Primarily we explain the data structure, the construction algorithm, and the implemented storage method. In addition, we address running time, memory usage and scaling behavior of PanTools. We end by illustrating possible applications that can be developed using our pan-genome representation as a foundation.

2 Methods

2.1 Overview

In De Bruijn graphs (DBGs), nodes correspond to unique k-mers (words) in the input sequences, and edges connect nodes whose words overlap by k – 1 nucleotides. Storing only one copy of each word, DBGs efficiently compress the input sequences. However, the precision in detecting sequence similarities depends on the choice of k. DBGs have been effectively employed for sequence applications such as de novo assembly (Pevzner et al., 2001; Zerbino and Birney, 2008), de novo repeat classification (Pevzner et al., 2004), genotyping (Iqbal et al., 2012), syntenic block finding (Minkin et al., 2013) and, more recently, for pan-genome representation (Baier et al., 2015; Beller and Ohlebusch, 2015; Holley et al., 2016; Marcus et al., 2014).

As in some other existing approaches, the core of our pan-genome is a compressed DBG, which is generalized through a number of key properties to make it efficient and applicable to real data. It is:

1. **Compressed**, to preserve space and allow efficient traversal. Compressing non-branching paths also improves the interpretability of the topology which is important for mining structures in the graph, for example, using the graph database query language, Cypher (Van Bruggen, 2014).

2. **Bi-directed**, to allow reverse complement sequences to be stored in the same nodes of the graph. This property slightly reduces the size of the graph, but most importantly makes it applicable to double-stranded data, for instance to detect inversions between genomes.

3. **Localized**, to store the genomic positions at which each forward or reverse sequence of a node appears.

4. **Indexed**, by all canonical k-mers of the dataset, to provide quick access to the node where a given k-mer occurs. A k-mer is called canonical if it is smaller than its reverse complement, lexicographically.

5. **General**, to allow storing ambiguous genomic regions in the pan-genome, which are widespread in real datasets.

The most straightforward way of constructing a compressed DBG is to build the original uncompressed DBG and compress non-branching paths. However, branch compression in a graph with billions of nodes requires an amount of memory only found in high-end machines. Therefore, efficient construction methods like those proposed in Beller and Ohlebusch (2015) and Baier et al. (2015) create a compressed DBG directly from input sequences. Below, we describe an alternative direct method for construction of the compressed DBG which is optimized for disk-based storage in a graph database. It is an online algorithm, i.e. the graph is updated as soon as the next k-mer of the input is scanned. This property enables us to cumulatively add new genomes to the pan-genome over time.

2.2 Data structures

The pan-genome is represented as a bi-directed graph, with two-sided nodes [forward (F) and reverse (R)] corresponding to the nucleotide sequence and its reverse complement. As a result, there are four types of edges (FF, FR, RF and RR) depending on the sides that are connected. Figure 1 illustrates a pan-genome graph (k = 3) of two genomes, each with one sequence, white and black, with shared parts colored gray. In this picture, circles, called **sequence nodes**, locate the nodes of the pan-genome where a sequence starts and ends, rectangles represent **normal nodes** and the rounded rectangle is an instance of a so-called **degenerate node** representing the ambiguous region in the first sequence. Ambiguous regions are sub-sequences in which all consecutive k-mers contain one or more ambiguous bases. In this example, the first sequence contains an R (a purine, i.e. an A or G), which has been stored in degenerate node 2. Degenerate nodes save the connectivity of the paths that input sequences take through the graph and facilitate the reconstruction of the constituent genomes or genomic regions. For simplicity, from here on we use the term ‘node’ as shorthand for normal nodes. In the current implementation, we also use some other types of nodes, which will be introduced in the Section 3. In Figure 1, all edges are of type FF except for the loop over node 4 and the edge between nodes 10 and 1, which both are of type FR (determined by the orientation of the arrows on the edge).

Table 1 lists the coordinates stored in each node. A coordinate determines the genomes where the sequence of the node occurs, the position at which it occurs, and whether the forward or reverse sequence occurs. As each genome may contain several sequences (chromosomes, contigs etc.) coordinates are represented by three numbers: genome, sequence and position. For example in node 1, the forward sequence, AAA, occurs in genome 1, sequence 1 at positions 0 and 1, and its reverse sequence, TTT, occurs in genome 2, sequence 1 at position 8.

![Fig. 1. A pan-genome graph (k = 3) for two sequences, AAAARATAATCCGCG (in white) and CATACTCCCTT (in black). Shared nodes are gray. Rectangle: normal node, rounded rectangle: degenerate node, circle: sequence node.](https://academic.oup.com/bioinformatics/article-abstract/32/17/i487/2450785)
DBGs have proven useful to visualize short variations between sequences, as they create various known substructures in the pan-genome graph, a fact well-known from (co-)assembly (Iqbal et al., 2012; Nijkamp et al., 2013; Pevzner et al., 2001). For example in Figure 1, the shared sequence ATA(C/A)TCC, with one SNP in the middle, forms a bubble in the center of the graph (induced by nodes 3, 6, 7 and 8). Moreover, the bi-directed pan-genome allows for detecting inversions. For instance, the subsequence TTT at the end of the second sequence can be detected as an inverted translocation of AAA at the start of the first sequence, by comparing the orientations of the coordinates stored in node 1.

Locating $k$-mers within the pan-genome is an essential operation for construction of the pan-genome and facilitates applications such as sequence retrieval, read mapping and sequence alignment. For these reasons, the pan-genome graph is accompanied by an ordered $k$-mer index which quickly locates each canonical $k$-mer by giving the node number, format (canonical or non-canonical) and relative position of its occurrence in the node (note that each $k$-mer occurs just once in the graph). The format of the $k$-mer at its first visit is stored and used by the construction algorithm. Table 2 shows the $k$-mer index of the example graph in Figure 1. Positions are expressed left to right, starting at 0.

### 2.3 Online construction of the pan-genome graph

We employ KMC2 (Deorowicz et al., 2014) to build a $k$-mer index. Our construction algorithm (Algorithm 1) is then based on four elementary operations: Create, Extend, Follow and Split. In the main algorithm, sequences are scanned in turn, and each subsequent $k$-mer is looked up in the index. If a $k$-mer is not visited yet in either orientation, a new node of length $k$ is added to the graph by the function Create, which is then Extended until we encounter a previously encountered $k$-mer. We locate a previously visited $k$-mer in the graph and, depending on the orientation in which it appears in a node, one of the Follow-forward or Follow-reverse operations will be performed. Both of these operations take advantage of the Split function to divide a node into two, taking care of the bookkeeping.

Figure 2b illustrates how these four simple operations work when the first five 3-mers of the black sequence (CATACTCCTTT) are added to the pan-genome of the white sequence (Fig. 2a). The first 3-mer, CAT, creates node 5 as it is not available in the white sequence. ATA appears in node 3, so it is followed to reach the next 3-mer of the node, TAA, which differs from the next 3-mer in the black sequence, TAC. Node 3 is therefore split, node 7 is initialized with TAC and is extended with ACT and CTC as these two 3-mers have not been visited before. Continuing this process, the next 3-mer, TCC, appearing in the middle of the node 6, results in another split to enter this node.

### 2.4 Choice of $k$

$k$-mers are the building blocks of every DBG, determining to a large extent its properties. Chikhi and Medvedev (2014) describe that in

| Node | Sequence | Coordinates | Node | Sequence | Coordinates |
|------|----------|-------------|------|----------|-------------|
| 1    | AAA      | F:1:1:0; R:2:1:8 | 6    | TAATC    | F:1:1:6    |
| 2    | AARAT    | F:1:1:2     | 7    | TACTC    | F:2:1:2    |
| 3    | ATA      | F:1:1:15; F:2:1:1 | 8    | TGGC     | F:1:1:9; F:2:1:5 |
| 4    | CGC      | F:1:1:11; R:1:1:12 | 9    | CGG      | F:1:1:10  |
| 5    | CAT      | F:2:1:0     | 10   | CCTT     | F:2:1:6    |

Table 1. Node properties for the graph in Figure 1

| k-mer | Pointer | k-mer | Pointer | k-mer | Pointer | k-mer | Pointer |
|-------|---------|-------|---------|-------|---------|-------|---------|
| AAA   | 1:C:0   | ATG   | 5:N:0   | AGG   | 10:N:0  | GTA   | 7:N:0   |
| AAG   | 10:N:1  | CGG   | 9:C:0   | ATA   | 3:C:0   | TAA   | 6:C:0   |
| AAT   | 6:C:1   | CGC   | 4:C:0   | ATC   | 6:C:2   |
| ACT   | 7:C:1   | CTC   | 7:C:2   | GGA   | 8:N:0   |

Table 2. The k-mer index for the graph in Figure 1

Pointers are given as (node:format:position). For format, C = canonical, N = non-canonical.

Algorithm 1. Pseudo-code of the construction algorithm.

```plaintext
Data: one or more Genomes each containing one or more Sequences, k-mer length k
Result: a pan-genome Graph and k-mer Index
Initialize empty Graph, build Index of all canonical k-mers;
for g = 1 .. number of Genomes do
  for s = 1 .. number of Sequences in g do
    position = 0;
    while position < length(Genomes[g].Sequences[s]): k+1 do
      kmer = Genomes[g].Sequences[s][position .. position + k – 1];
      if kmer is visited for the first time then
        n = Create(kmer);
        Extend(n);
      else
        n = node where kmer occurs;
        if kmer visited in this orientation then
          Follow-forward(n, kmer);
        else
          Follow-reverse(n, kmer);
        end
      end
    end
  end
end
```

Fig. 2. Four basic operations of the construction algorithm. (a) Pan-genome of sequence AAAARATAATCCCGC. (b) Adding the first five 3-mers of the sequence CATACTCCTTT to the pan-genome of AAAARATAATCCCGC, results in creating node 5, splitting node 3 and creating and extending node 7.
DBG-based assemblers, the best choice of \( k \) is the one that provides the largest number of distinct, non-erroneous genomic \( k \)-mers to the assembler. When using a (compressed) DBG to represent a pan-genome graph, although it is not easy to define an optimal value for \( k \), it is possible to suggest a lower bound to avoid tangling the graph. Short \( k \)-mers increase the chance of single nodes representing unrelated subsequences within and between different genomes, making the graph tangled and hard to interpret. On the other hand, selecting a large value for \( k \) will decrease connectivity and lead to distinct node sets representing each genome or at least will obscure small-scale variation, e.g. SNPs less than \( 100 \) \text{bp}.

### 2.5 Implementation

A major goal of our pan-genome project is to allow storage and exploration of variable pan-genomes for large collections of crop genomes, for example maize, rice and tomato (Álvitos et al., 2014; Chia et al., 2012; Li et al., 2014). To achieve this goal, the pan-genome graph and accompanying data structures are not maintained in memory, but in memory-mapped databases. The advantage is that the operating system takes care of reading and writing required chunks of files (pages) and the application just interacts with memory, which results in very fast I/O operations. Furthermore, memory mapped databases can be shared between different processes, which paves the way for developing multi-threaded pan-genomic applications in future. It should be noted that interacting with large files using disproportionately small amounts of memory increases the number of page faults, drastically reducing performance. Also, like any other disk-based program, the performance of PanTools depends on disk speed. Thus, to achieve the best performance we suggest to have a dedicated machine, preferably with a RAM drive or a solid-state drive (SSD).

We use three memory mapped databases: the Neo4j graph database, the index database and the genome database. The graph database contains information about nodes, relationships (edges) and their properties. The index database is a set of files representing the \( k \)-mer index, and the genome database, which is only used during the construction of the pan-genome, stores the compressed input sequences. The only data structure which is kept in memory is the small database of prefixes produced by KMC2. As a result, the memory requirement of the construction algorithm (and any other application which needs the index database) stays independent of the size of data. This enables us to create and explore graphs with millions of nodes using a constant, limited amount of memory, say 8GB.

### 3 Results

In this section, we demonstrate the functionality and performance of our pan-genome approach, and discuss its application in comparative genomics. To this end, we constructed pan-genomes of two HIV-1 strains (AF069671.1 and AF413987.1), 62 Escherichia coli genomes (Marcus et al., 2014), 93 yeast genomes (Strope et al., 2015) and 19 Arabidopsis thaliana genomes (Gan et al., 2011). Our experiments were conducted on a Linux server (Ubuntu 14.04) with an Intel® Xeon® X5660@2.8GHz, with 24 logical cores, 64GB RAM and a 32GB RAM disk.

#### 3.1 Functionality

The current implementation of PanTools provides the following functionality:

1. **Build**, given a number of FASTA files, constructs the pan-genome.
2. **Add**, adds one or more new genomes to a given pan-genome.
3. **Annotate**, given one or more GFF files, adds gene nodes to the graph corresponding to the annotated genes.
4. **Group**, adds group nodes to the graph linking genes by some criterion, for instance orthology or name.
5. **Retrieve**, extracts the sequence of specified genes or genomic regions from the pan-genome graph.
6. **Reconstruct**, reconstructs some/all of the constituent genomes from a given pan-genome.
7. **Compare**, compares the topology of two existing pan-genomes.
8. **Query**, gives a command prompt to run Cypher queries and receive the results.

![Fig. 3. The number of non-unique \( k \)-mers in a set of 100M random \( k \)-mers, as a function of \( k \)]
To demonstrate the **Build**, **Annotate** and **Group** functionality, we constructed the pan-genome of two HIV-1 strains, annotated the genomes and grouped the homologous genes. Figure 4 visualizes the graph, with different types of nodes indicated by different colors explained in the caption. The **pan-genome node** points to two genome nodes (1 and 2) each containing a single sequence (1.1 and 2.1) (Fig. 4a). Two instances of *gag* genes of equal length (246 nt) have been grouped together by a *gag* group node (Fig. 4b). One of these genes begins and ends at a node of length 437, the other at another node of length 443. The exact position where each gene starts and end is clear in Figure 4c that the two homologous *pol* genes have different lengths, 3012 and 3006, and begin and end in distinct nodes. The pair of nodes where these genes begin and end belong to a bubble, which indicates that there is some variation (indel or SNP) between them. The *pol* genes are rather variable, as they are represented by the entire chain of bubbles at the top-right of the graph. SNPs can be distinguished based on the fact that the length of both branching nodes equals $2k-1$ (here 63), which means nodes differ in a single nucleotide in the middle of their sequence (Fig. 4d). Each edge has a three-letter label, starting with the two nucleotides which appear at the borders of the $k-1$ overlap of the nodes and ending with a number in the range $0-3$ which codes for the four types of edges. The incoming and outgoing edges of the top node have been labeled CA0 and AG0, respectively, while those of the node at the bottom are labeled CG0 and GG0, indicating a G-A SNP.

Our method allows to incrementally adding new genomes to an existing pan-genome. To show this, we constructed a pan-genome of three yeast genomes and iteratively added sets of 10 new yeast genomes. Then, we compared the nine intermediate pan-genomes with those constructed directly from 13, ..., 93 genomes from scratch (called Y13 to Y93). Using the **Compare** functionality, we observed that pan-genomes containing the same genomes but constructed in different ways (directly or iteratively) were identical, having the same number of $k$-mers, nodes, edges and bases as well as the same properties stored in corresponding nodes. We also verified that changing the order of the genomes in the input dataset does not affect the construction, resulting in isomorphic pan-genomes.

### 3.2 Performance

To verify the resource requirements of the construction algorithm when it scales to larger datasets, we constructed nine *A. thaliana* pan-genomes containing $3, 5, \ldots, 19$ genomes. Table 3 reports properties of the resulting pan-genomes, as well as the resources consumed during the construction. In all experiments, we set $k = 31$.

PanTools created the pan-genome of 19 *A. thaliana* genomes in less than an hour using just above 6 GB of memory. To verify that our method scales to arbitrary genome sizes, we also successfully constructed a pan-genome graph of seven human genomes in a preliminary experiment.

The numbers of $k$-mers, nodes and edges in the pan-genome grow in a sub-linear fashion with respect to the size of the datasets. The ratio of the number of $k$-mers to the number of nodes in the Arabidopsis pan-genomes ranges between 22 and 49, which indicates the significant effect of compressing the non-branching paths. However, this ratio declines as more genomes are added to the pan-genome.

Pan-genomes store far fewer base pairs than linear genomes because redundancy is eliminated. Table 3 also shows that the number of base pairs stored in the pan-genome representation of 19 *A. thaliana* genomes (A19) is just under one fourth of those stored in the linear genomes. This difference becomes more significant as more genomes are added to the pan-genome.

To compare our construction method with the existing in-memory methods we run the FM-index based algorithm (FMI) presented by Beller and Ohlebusch (2015) and the BWT-based
on 29 July 2018

Table 5. Average elapsed time (in milliseconds) for annotating one gene, retrieving one gene, retrieving 1 kbp and reconstructing one genome in five different yeast pan-genomes

| Gene annotation | Gene retrieval | 1 kbp retrieval | Genomic reconstruction |
|-----------------|----------------|-----------------|------------------------|
| Y13             | 2.2            | 0.3             | 2.3                    | 3615                    |
| Y33             | 3.5            | 0.5             | 3.4                    | 7453                    |
| Y53             | 4.8            | 0.8             | 4.6                    | 11 639                  |
| Y73             | 6.2            | 1.2             | 6.6                    | 15 236                  |
| Y93             | 7.6            | 1.6             | 7.9                    | 19 544                  |

algorithm presented by Baier et al. (2015), on datasets of 62 E. coli (E62), 93 yeast (Y93) and 19 A. thaliana (A19) genomes; results are presented in Table 4. We also tried to run SplitMEM (Marcus et al., 2014), but it ran out of memory even on the smallest dataset in this experiment. The DBGs produced by the FMI- and BWT-based tools are exactly identical and these tools show the same performance. However, their graph contains more nodes than the graph produced by PanTools. The reason is that they do not build a bi-directed DBG and reverse complement sequences are stored in different nodes. The smaller number of edges in our pan-genome is explained by the fact that we store each edge only once, whereas the two other tools store each individual transition. To conclude, besides the various useful features that our disk-based approach provides, the amounts of time and memory it needs are comparable to the best in-memory methods, in particular for larger datasets.

Efficient extraction of genes, genomic regions and genomes from a pan-genome is a key functionality for downstream pan-genome applications. A genomic feature is extracted by traversing the path it takes in the pan-genome, which is determined using the genomic positions of each node, stored as node properties. To examine annotation and retrieval efficiency, we annotated and then retrieved all genes, extracted thousand 1000 nt-long random genomic regions and reconstructed the whole set of constituent genomes in five different yeast pan-genomes; Table 5 gives the run-times in milliseconds. PanTools retrieves a gene in around one millisecond. As expected, the average annotation and retrieval times increase slightly as the pan-genome grows.

3.3 Pan-genome applications

Expecting an increasing rate of genome production, pan-genomes should ideally take over the role of linear reference genomes in comparative genomics. This implies that in the future, we will analyze novel genomic data with respect to all genomes in the pan-genome at once, move from pairwise genome comparisons to multiple genome comparisons at once, and browse pan-genomes rather than reference genomes. Another important application is variation detection in pan-genomes, including single-nucleotide polymorphisms, structural variation, copy-number variation, synteny, transposon-insertion polymorphisms, etc. A pre-requisite for such high-level applications is a solid data structure and construction algorithm, as presented in this article, possibly enhanced with application-specific indices.

A basic application, underlying genome browsing and several analyses, is fast retrieval of genes and genomic regions of interest and assess the variations. To demonstrate this feature, we retrieved all instances of the well-known FRIGIDA gene from the pan-genome of 19 A. thaliana genomes (A19). This gene encodes a major determinant of natural variation in flowering time and its allelic diversity among these 19 accessions is of interest to plant biologists (Gan et al., 2011). There is significant allelic variation in this gene, as shown in Figure 5. The variants differ at the start of the gene, but are identical in the last 106 nucleotides. We can distinguish types of variants by assessing the ‘bubbles’ in the graph: the 11 bubbles correspond to 6 SNPs, 1 deletion and 4 mismatches less than k nucleotides apart. The exact presentation of genomic variations like these needs to be worked out in more detail, but the example illustrates that we can quickly assess genomic variation in multiple genomes at once, rather than compose the full picture from many pairwise genome comparisons.

Many pan-genome analyses will require searches in the highly connected graph database. We will use the query language of Neo4j, Cypher, to search pan-genomes for specific structures and properties. The efficiency of the queries depends on how constrained the query is. Table 6 presents the result of some queries from a pan-
genome of three yeast genomes. The first three queries ask for the number of bubbles created by two branching paths with at most 2 (simple bubble), 3 or 4 edges, respectively. As expected, the runtime of such queries increases as the length of the branching path increases. The fourth query returns all sequences that belong to simple SNPs (one node in each branch). The fifth query gives the sequence of two branches of simple bubbles where one branch is a degenerate node. The sixth query returns the occurrence arrays of nodes shared by chromosome 1 of all the genomes, and the seventh one gives the number of nodes that are specific to chromosome 1 of genome 1. These examples show that Cypher supports many different queries, which can be done in reasonable time. Users can run these queries on their own pan-genome using the query command of PanTools.

4 Conclusion

Thanks to large sequencing efforts, many species or phylogenetic clades are no longer represented by a single reference genome, but by a multitude of genomes. Besides the sequence similarities between related genomes, there may be significant variation in genomic content and organization, which was often the reason to sequence and study them. To deal with this new data challenge, there is an increasing need for new ways of storing and constructing unified representations of large collections of genomes. In addition, we need accompanying algorithms to answer key questions, such as on core and dispensable genes, recurring genetic variants and structural variation, which are cumbersome to address using linear representations of large numbers of genomes. In this article, we have presented PanTools, an implementation of a pan-genome representation based on the Neo4j graph database, focusing on the application to large sets of complex eukaryotic genomes. The program allows for the construction of pan-genome databases of many genomes, and contains extensions such as adding sequences, genes and orthology annotations, using relatively modest computational resources.

PanTools offers a good starting point for developing various pan-genomic applications, such as multi-genome read mapping, pan-genome exploration (visualization, browsing), structure-based variation detection and comparative genomics. To efficiently implement algorithms supporting such analyses it is likely that additional layers of annotation, summaries (e.g. synten blocks) or different indices will be needed. The current base implementation in Neo4j is adaptable and extensible and offers an excellent foundation for such extensions. In summary, we have presented a first implementation of a pan-genome representation and construction algorithm, which can form the basis of a collection of tools to allow pan-genomes to take over the role of linear reference genomes in genomics.

Acknowledgements

We thank Maria-Anna Misiakou and Salvador Casani Galdo for valuable input.

Funding

This work has been published as a part of the research project called Pan-genomics for crops funded by the Graduate School Experimental Plant Sciences (EPS) in the Netherlands.

Conflict of Interest: none declared.

References

Afinoz, S. et al. (2014) Exploring genetic variation in the tomato (Solanum section Lycopersicon) clad by whole-genome sequencing. Plant J., 80, 136–148.
Bier, U. et al. (2015) Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics, 32, 497–504.
Beller, T. and Ohlebusch, E. (2015). Efficient construction of a compressed de Bruijn graph for pan-genome analysis. In: Lecture Notes in Computer Science, volume 9133, Springer Verlag, Berlin, Germany, pp. 40–51.
Chia, J. M. et al. (2012) Maize HapMap2 identifies extant variation from a genome in flux. Nat. Genet., 44, 803–807.
Chikhi, R. and Medvedev, P. (2014) Informed and automated k-mer size selection for genome assembly. Bioinformatics, 30, 31–37.
Deorowicz, S. et al. (2014) KMC 2: fast and resource-frugal k-mer counting. Bioinformatics, 31, 1569–1576.
Fleischmann, R. D. et al. (1995) Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. Science, 269, 496–512.
Fraser, C. M. et al. (1995) The minimal gene complement of Mycoplasma genitalium. Science, 270, 397–403.
Gan, X. et al. (2011) Multiple reference genomes and transcriptomes for Arabidopsis thaliana. Nature, 477, 419–423.
Have, C. T. and Jensen, L. J. (2013) Are graph databases ready for bioinformatics? Bioinformatics, 29, 3107–3108.
Holley, G. et al. (2016) Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. Algoritmia Mol. Biol., 11, 1–9.
Iqbal, Z. et al. (2012) De novo assembly and genotyping of variants using color-coded de Bruijn graphs. Nat. Genet., 44, 226–232.
Li, J. Y. et al. (2014) The 3,000 rice genomes project: new opportunities and challenges for future rice research. GigaScience, 3, 1–3.
Marcus, S. et al. (2014) SplitMEM: a graphical algorithm for pan-genome analysis with suffix skaps. Bioinformatics, 30, 3476–3483.
Minkin, I. et al. (2013). Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In Lecture Notes in Computer Science, volume 8126 LNBI, Springer, Berlin, pp. 215–229.
Nijakamp, J. F. et al. (2013) Exploring variation-aware contig graphs for (comparative) metagenomics using MaryGold. Bioinformatics, 29, 2826–2834.
Pevzner, P. A. et al. (2001) An Eulerian path approach to DNA fragment assembly. Proc. Natl. Acad. Sci. USA, 98, 9748–9753.
Pevzner, P. A. et al. (2004) De novo repeat classification and fragment assembly. Genome Res., 14, 1786–1796.
Robinson, L. et al. (2013). Graph Databases. O’Reilly, Sebastopol, CA.
Strope, P. K. et al. (2015) The 100-genomes strains, an S. cerevisiae resource that illuminates its natural phenotypic and genotypic variation and emergence as an opportunistic pathogen. Genome Res., 125, 762–774.
Tettelin, H. et al. (2005) Genome analysis of multiple pathogenic isolates of Streptococcus agalactiae: implications for the microbial “pan-genome”. Proc. Natl. Acad. Sci. USA, 102, 13950–13953.
Van Bruggen, R. (2014). Learning Neo4j. Packt Publishing Ltd, Birmingham, UK.
Zerbino, D. R. and Birney, E. (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res., 18, 821–829.