Research Article

Kadir Dede and Enno Ohlebusch*

Dynamic construction of pan-genome subgraphs

<https://doi.org/10.1515/comp-2020-0018>

Received Oct 21, 2019; accepted Feb 25, 2020

Abstract: Marcus et al. (Bioinformatics 2014) proposed to use a compressed de Bruijn graph as a description of a pan-genome, comprising the genomes of many individuals/strains of the same or closely related species. Subsequent work improved the construction of the compressed de Bruijn graph in terms of run-time and memory consumption. According to the Computational Pan-Genomics Consortium (Briefings in Bioinformatics 2016), a pan-genome data structure should support the following functionality: “All information within a data structure should be easily accessible for human eyes by visualization support on different scales.” However, a pan-genome graph can have thousands to millions of nodes and such an amount of information is certainly not easily accessible for human eyes. Thus, the possibility to construct pan-genome subgraphs on demand would be quite valuable. In this article, we use the space-efficient representation of the compressed de Bruijn graph devised by Beller and Ohlebusch (Algorithms for Molecular Biology 2016) to construct pan-genome subgraphs on the fly. The user can specify a region in one of the genomes and the software tool will build a subgraph that contains the path corresponding to that region and all paths that are in the neighborhood of that path. The size of the neighborhood can be controlled by the user.

Keywords: compressed de Bruijn graph, Burrows-Wheeler transform, backward search, pan-genome analysis

1 Introduction

Tettelin et al. [26] coined the term pan-genome a decade ago; they evaluated the composition of 6 strains of *Streptococcus agalactiae*. In a broader sense, the pan-genome defines the entire genomic repertoire of a given phylo-

genetic clade (which may range from species to phylum and beyond). One must distinguish between gene-based and sequence-based methods to analyze a pan-genome. In the gene-based approach, one distinguishes between the core genome that contains genes shared by all strains within the clade (housekeeping genes etc.), the dispensable genome (made of genes shared by only a subset of the strains), and strain-specific genes. In this article, we use a sequence-based approach and the term ‘pan-genome’ refers to the whole genomic content of a certain species or population.

We next recall the definition of de Bruijn graphs because they play a key role in this article. Given a set of strings S and a natural number k , the de Bruijn graph representation of S contains a node for each distinct k -mer (length k string) observed in S and a directed edge $u \rightarrow v$ for every observed $(k+1)$ -mer in S for which the first k nucleotides coincide with u and the last k nucleotides coincide with v ; see Figure 1 for an example.

In our application, a k -mer and its reverse complement are not represented by the same node because we use single strands (i.e., the graph is uni-directional). In a very different application, de Bruijn graphs are used to assemble genomes from short read data. In genome assembly, a bi-directed graph representation is required because it is not known a priori from which strand a read originated.

The concept of a de Bruijn graph was extended by Iqbal et al. [15]. They introduced the colored de Bruijn graph, which is aimed at “detecting and genotyping simple and complex genetic variants in an individual or population.” Now each input string S^i is associated with a color c_i and if a node (k -mer) or an edge occurs in S^i , then it is colored with c_i . Consequently, a node or an edge can have multiple colors (if we allow multiple edges between two nodes, then each edge has a unique color). There are two possibilities to efficiently store such colored de Bruijn graphs: Bloom filter tries [14] or succinct representations [20]. A (colored) de Bruijn graph is a node-labeled directed graph, but (colored) edge-labeled directed graphs are also possible; see e.g. the population reference graph introduced by Dilthey et al. [7].

Kadir Dede: Institute of Theoretical Computer Science, Ulm University, D-89069 Ulm, Germany; Email: kadir.dede@gmx.net

***Corresponding Author: Enno Ohlebusch:** Institute of Theoretical Computer Science, Ulm University, D-89069 Ulm, Germany; Email: Enno.Ohlebusch@uni-ulm.de

Figure 1: The de Bruijn graph with multiple edges of the three strings ACGAATCACCAA, ACGAATCAGCAA, and GCGAATCTTTCTTTTCAA for $k = 3$.

Figure 2: The compressed de Bruijn graph of the three strings ACGAATCACCAA, ACGAATCAGCAA, and GCGAATCTTTCTTTTCAA for $k = 3$. Each node has a unique identifier; the calculation of identifiers will be explained in Section 2.3.

Marcus et al. [19] proposed to use a compressed¹ de Bruijn graph for representing an entire population of genomes. Ideally, it takes multiple assembled genomes as input, but it can also work with contigs. In their approach, there may be multiple edges between two nodes u and v : whenever the k -mers u and v occur consecutively in a string S^c , i.e., $u = S^c[i..i+k-1]$ and $v = S^c[i+1..i+k]$,² there is a directed edge $u \rightarrow v$, which is stored in an adjacency list. Moreover, the pair (c, i) consisting of the color c of the string S^c and the starting position i of u in S^c is stored at node u . Such a de Bruijn graph can be compressed by merging non-branching chains of nodes into a single node, which is labeled with a longer string. More precisely, if node u is the only predecessor of node v and v is the only successor of u (but there may be multiple edges $u \rightarrow v$), then u and v can be merged into a single node that has the predecessors of u and the successors of v . After

maximally compressing the graph, every node (apart from possibly the start nodes) has at least two different predecessors or its single predecessor has at least two different successors and every node (apart from the stop nodes) has at least two different predecessors; see Figure 2 for an example and [1, 19] for details.

Of course, the compressed de Bruijn graph can be built from its uncompressed counterpart (a much larger graph), but this is disadvantageous because of the huge space consumption. That is why Marcus et al. [19] proposed to build it directly. Technically, they work with one string S that is obtained by concatenating the genomes with separator symbols in between. In the resulting compressed de Bruijn graph, each node corresponds to a substring ω of S and consists of the four components $(id, len, posList, adjList)$, where id is a natural number that uniquely identifies the node, len is the length $|\omega|$ of ω , $posList$ is the list of positions at which ω occurs in S (sorted in ascending order), and $adjList$ is the list of the successors of the node. Moreover, the list $adjList$ is sorted in such a way that the path through the graph G that gives

¹ The term ‘compact’ seems to be more appropriate, but we will use the name coined by Marcus et al. [19].

² $S[i..j]$ denotes the substring of string S starting with the character at position i and ending with the character at position j .

S is induced by the adjacency lists: if node $G[id]$ is visited for the i -th time, then its successor is the node that can be found at position i in the adjacency list of $G[id]$. Baier et al. [1] presented algorithms that construct the compressed de Bruijn graph faster and using significantly less space (two orders of magnitude) than splitMEM [19]. Beller and Ohlebusch [3] were able to further improve the approach based on the BWT by proposing a new space-efficient implicit representation of the compressed de Bruijn graph that adds the possibility to search for a pattern within the pan-genome. The algorithm that constructs the implicit representation uses a full text index in form of the wavelet tree of the Burrows-Wheeler transform of S . This full text index supports backward search and is independent of the value of k . Thus, if one wants to build the compressed de Bruijn graph of S for a different value of k , it is not necessary to rebuild the index. For a fixed k , the implicit representation is constructed as follows: two bit-vectors are computed with the help of the index structure and then the nodes of the compressed de Bruijn graph are calculated and stored in an array G . Each node has an identifier that can be computed from the bit-vectors, and the node with identifier id is stored at $G[id]$. As explained above, a node $G[id]$ corresponds to a string ω . In the representation, we have $G[id] = (len, lb, size)$, where len is the length of ω , lb is the left boundary and $size$ is the size of the suffix array interval of ω (hence its right boundary is $rb = lb + size - 1$). To explain the notion ‘suffix array interval’, let us recall that the suffix array of a string S is an integer array SA so that $SA[i]$ is the starting position of the i -th lexicographically smallest suffix of S , where $1 \leq i \leq |S|$. The suffix array interval of a substring ω of S is the largest interval in SA in which all suffixes start with ω . It should be stressed that this representation of the compressed de Bruijn graph is implicit because (a) it does not store the edges of the graph and (b) at each node it does not store $posList$, but only the left boundary lb and the $size$ of a suffix array interval. The list of positions $posList$ can be obtained as $[SA[j] \mid lb \leq j \leq rb]$ if necessary, where $rb = lb + size - 1$.

According to the Computational Pan-Genomics Consortium [27], a pan-genome data structure should support the following functionality: “All information within a data structure should be easily accessible for human eyes by visualization support on different scales.” However, a pan-genome graph can have thousands to millions of nodes and such an amount of information is certainly not easily accessible for human eyes. Thus, we investigated if it is possible to construct pan-genome subgraphs on demand.

2 Methods

As already mentioned, we use the space-efficient representation of the compressed de Bruijn graph devised by [3] to construct pan-genome subgraphs on the fly. The user can specify a region in one of the genomes and the software tool will build a subgraph that contains the path corresponding to that region (the *reference path*) and its d -neighborhood, i.e., all nodes that have a distance $\leq d$ to a node in the reference path.

2.1 Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the sentinel character $\$$. In the following, S is a string of length n on Σ having the sentinel character at the end (and nowhere else). In pan-genome analysis, S is the concatenation of multiple genomic sequences, where the different sequences are separated by special symbols.

For $1 \leq i \leq n$, $S[i]$ denotes the *character at position i* in S . For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j . Furthermore, S_i denotes the i -th suffix $S[i..n]$ of S . The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S , that is, it satisfies $S_{SA[1]} < S_{SA[2]} < \dots < S_{SA[n]}$; see Table 1 for an example. A suffix array can be constructed in linear time; see e.g. the overview article [23]. For every substring ω of S , the ω -interval is the suffix array interval $[i..j]$ so that ω is a prefix of $S_{SA[k]}$ if and only if $i \leq k \leq j$.

The Burrows-Wheeler transform [5] converts S into the string $BWT[1..n]$ defined by $BWT[i] = S[SA[i] - 1]$ for all i with $SA[i] \neq 1$ and $BWT[i] = \$$ otherwise; see Table 1. Several semi-external and external memory algorithms are known that construct the BWT directly (i.e., without constructing the suffix array); see e.g. [4, 8, 17, 22].

A bit vector B can be preprocessed in linear time so that rank-queries ($rank_b(B, i)$ asks for the number of occurrences of bit b in B up to and including position i) and select-queries ($select_b(B, i)$ asks for the position of the i -th occurrence of bit b in B) can be answered in constant time [6, 16].

The *wavelet tree* [12] of a string T supports rank-queries ($rank_c(T, i)$ asks for the number of occurrences of character c in T up to and including position i) and select-queries ($select_c(T, i)$ asks for the position of the i -th occurrence of character c in T) in $O(\log \sigma)$ time. This is possible because each node in the wavelet tree stores a bit vector in

Table 1: The suffix array SA of the string `ACGAATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA$` and related notions are defined in Section 2.1. The bit vectors B_r and B_l for $k = 3$ are explained in Section 2.3.

i	SA	BWT	B_r	B_l	LF	Ψ	$S_{SA[i]}$
1	45	A	0	0	4	14	\$
2	13	A	0	0	5	15	#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
3	26	A	0	0	6	35	#GCGAATCTTTCTTTTCAA\$
4	44	A	0	0	7	1	A\$
5	12	A	0	0	8	2	A#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
6	25	A	0	0	9	3	A#GCGAATCTTTCTTTTCAA\$
7	43	C	0	0	20	4	AA\$
8	11	C	0	0	21	5	AA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
9	24	C	0	0	22	6	AA#GCGAATCTTTCTTTTCAA\$
10	4	G	0	0	31	17	AATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
11	17	G	0	0	32	18	AATCAGCAA#GCGAATCTTTCTTTTCAA\$
12	30	G	0	0	33	19	AATCTTTCTTTTCAA\$
13	8	C	0	0	23	25	ACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
14	1	\$	0	0	1	26	ACGAATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
15	14	#	0	1	2	27	ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
16	21	C	0	0	24	34	AGCAA#GCGAATCTTTCTTTTCAA\$
17	5	A	1	0	10	37	ATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
18	18	A	0	0	11	38	ATCAGCAA#GCGAATCTTTCTTTTCAA\$
19	31	A	1	0	12	39	ATCTTTCTTTTCAA\$
20	42	T	1	0	36	7	CAA\$
21	10	C	0	0	25	8	CAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
22	23	G	1	0	34	9	CAA#GCGAATCTTTCTTTTCAA\$
23	7	T	0	0	37	13	CACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
24	20	T	0	0	38	16	CAGCAA#GCGAATCTTTCTTTTCAA\$
25	9	A	0	1	13	21	CCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
26	2	A	0	0	14	31	CGAATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
27	15	A	0	0	15	32	CGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
28	28	G	0	0	35	33	CGAATCTTTCTTTTCAA\$
29	33	T	0	0	39	44	CTTTCTTTTCAA\$
30	37	T	0	1	40	45	CTTTTCAA\$
31	3	C	0	0	26	10	GAATCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
32	16	C	0	0	27	11	GAATCAGCAA#GCGAATCTTTCTTTTCAA\$
33	29	C	0	0	28	12	GAATCTTTCTTTTCAA\$
34	22	A	0	1	16	22	GCAA#GCGAATCTTTCTTTTCAA\$
35	27	#	0	1	3	28	GCGAATCTTTCTTTTCAA\$
36	41	T	1	0	41	20	TCAA\$
37	6	A	0	0	17	23	TCACCAA#ACGAATCAGCAA#GCGAATCTTTCTTTTCAA\$
38	19	A	1	0	18	24	TCAGCAA#GCGAATCTTTCTTTTCAA\$
39	32	A	0	0	19	29	TCTTTCTTTTCAA\$
40	36	T	0	0	42	30	TCTTTTCAA\$
41	40	T	1	0	43	36	TTCAA\$
42	35	T	1	0	44	40	TTCTTTTCAA\$
43	39	T	1	0	45	41	TTTCAA\$
44	34	C	0	0	29	42	TTTCTTTTCAA\$
45	38	C	1	0	30	43	TTTTCAA\$

such a way that rank- and select-queries can be answered in constant time and the height of a wavelet tree is $O(\log \sigma)$.

The LF-mapping (last-to-first-mapping) is defined as follows: if $SA[i] = q$, then $LF(i)$ is the index j so that $SA[j] = q - 1$ (if $SA[i] = 1$, then $LF(i) = 1$). In other words, if the i -th entry in the suffix array is the suffix S_q , then $LF(i)$ ‘points’ to the entry at which the suffix S_{q-1} can be found; see Table 1. The function Ψ is the inverse of the LF-mapping. Using the C -array (for each $c \in \Sigma$, $C[c]$ is the overall number of occurrences of characters in BWT that are strictly smaller than c) and the wavelet tree of the BWT, a value $LF(i)$ or $\Psi(i)$ can be calculated in $O(\log \sigma)$ time as follows:

$$LF(i) = C[c] + \text{rank}_c(\text{BWT}, i), \text{ where } c = \text{BWT}[i]$$

$$\Psi(i) = \text{select}_c(\text{BWT}, i - C[c]), \text{ where } c = S[SA[i]]$$

The wavelet tree of the BWT in combination with the C -array supports one backward search step in $O(\log \sigma)$ time [9]: given the ω -interval $[lb..rb]$ and a character c from the alphabet Σ , $\text{backwardSearch}(c, [lb..rb])$ returns the $c\omega$ -interval $[i..j]$ (i.e., $i \leq j$ if $c\omega$ is a substring of S ; otherwise $i > j$). Backward search can be generalized on the wavelet tree as follows: given an ω -interval $[lb..rb]$, a slight modification of the procedure $\text{getIntervals}([lb..rb])$ described in [2] returns the list $[(c, [i..j]) \mid c\omega \text{ is a substring of } S \text{ and } [i..j] \text{ is the } c\omega\text{-interval}]$, where the first component of an element $(c, [i..j])$ must be a character. The worst-case time complexity of the procedure getIntervals is $O(z + z \log(\sigma/z))$, where z is the number of elements in the output list; see [10, Lemma 3].

A substring ω of S is a *repeat* if it occurs at least twice in S . Let ω be a repeat of length ℓ and let $[i..j]$ be the ω -interval. The repeat ω is *left-maximal* if $|\{ \text{BWT}[x] \mid i \leq x \leq j \}| \geq 2$, i.e., the set $\{S[SA[x] - 1] \mid i \leq x \leq j\}$ of all characters that precede at least one of the suffixes $S_{SA[i]}, \dots, S_{SA[j]}$ is not singleton (where $S[0] := \$$). Analogously, the repeat ω is *right-maximal* if $|\{S[SA[x] + \ell] \mid i \leq x \leq j\}| \geq 2$. A left- and right-maximal repeat is called *maximal* repeat. (Note that [19] use the term “maximal exact match” instead of the more common term “maximal repeat”. We will not use the term “maximal exact match” here.) A detailed explanation of the techniques used here can be found in [21].

Given multiple genomic sequences $S^1, \dots, S^m$ with $S^i \neq S^j$ for $i \neq j$, we want to construct a compact index data structure for the m sequences in form of a multi string BWT. If we would construct the BWT of the string $S = S^1\$_1S^2\$_2 \dots S^m\$_m$, where $\$_1 < \$_2 < \dots \$_m$ are pairwise different separator symbols that are smaller than all other symbols occurring in the sequences, then the first m lexicographically smallest suffixes of S start with

$\$_1, \$_2, \dots, \$_m$ (in that order). In other words, the last character of the i -th input string S^i appears at the i -th position in the BWT of S (so S^i can be reconstructed from there). In practice, however, the use of pairwise different separator symbols would blow up the alphabet, which has a severe impact on the run-time. For this reason, we construct the BWT of the string $S = S^1\#S^2\# \dots S^{m-1}\#S^m\$$, where $\#$ is a special separator symbol that is smaller than all other symbols occurring in the sequences, but larger than $\$$. In this case, the last character of the m -th input string S^m appears at the first position of the BWT of S because $\$$ is the lexicographically smallest suffix of S . The position at which the last character of the i -th input string S^i appears, where $1 \leq i \leq m - 1$, depends on the lexicographic order of $S^2, S^3, \dots, S^m$. If S^{i+1} is the j -th smallest string among $S^2, S^3, \dots, S^m$, then the last character of S^i appears at position $j + 1$ of the BWT of S .

2.2 The overall picture: constructing a compressed de Bruijn subgraph on demand

In the introduction, we have defined uncompressed and compressed de Bruijn graphs. Figure 1 shows the uncompressed de Bruijn graph of the strings $S^1 = \text{ACGAATCACCAA}$, $S^2 = \text{ACGAATCAGCAA}$, and $S^3 = \text{GCGAATCTTTCTTTTCAA}$ for $k = 3$, while Figure 2 shows the corresponding compressed graph. However, we do not wish to construct the whole graph. By contrast, if the user specifies a region in one of the genomes, we will merely generate the corresponding path. In our example, for the region $S[7..12] = S^1[7..12] = \text{CACCAA}$, the reference path contains only the nodes 7 and 2. Moreover, we will construct the neighborhood of the reference path, which depends on the parameter d . The argument d determines the depth of the subgraph. Let u and v be nodes in a compressed de Bruijn graph. Then we define the distance $\text{dist}(u, v)$ to be the length³ of the shortest (directed) path from u to v . Let $R = \{v_1, \dots, v_l\}$ be the set of the nodes in the reference path. We define the compressed de Bruijn subgraph relative to R with depth d to be the compressed de Bruijn graph containing all nodes u satisfying $\text{dist}(u, v) \leq d$ for a node $v \in R$.⁴ It is a consequence of this definition that the de Bruijn subgraph relative to R is kind

³ The length of a path is the number of edges in the path. In particular, $\text{dist}(u, v)$ does not depend on the size of the nodes.

⁴ Note that in our implementation d must always be greater than or equal to 1.

Figure 3: The compressed de Bruijn subgraph of the three strings ACGAATCACAA, ACGAATCAGCAA, and GCGAATCTTTCTTTCAA for $k = 3$, $d = 2$, $start = 7$ and $end = 12$. The reference path is $7 \rightarrow 2$.

of asymmetrical. To see this, consider a node u in the de Bruijn graph.

1. If there is a path of suitable length from u to a node v in R , then u will be in the subgraph.
2. If there is a path of suitable length from a node v in R to u , then u will not be in the subgraph (unless case 1 applies).

We use this definition of subgraph because our construction algorithm is based on a backward search procedure.⁵

Figure 3 shows the subgraph in our example, where $R = \{2, 7\}$ and $d = 2$. For each node in the subgraph, we store the following components (see Figure 4 for an example):

- $node.id$, the unique identifier of the node
- $node.len$, length of the string ω corresponding to the node
- $node.lb$, left boundary of the ω -interval in the suffix array
- $node.size$, size of the ω -interval in the suffix array
- $node.suffix_lb$, left boundary of the interval of the k -mer suffix of ω
- $node.pred_list$, the list of all predecessors of the node in the subgraph

Note that the string corresponding to the node is $\omega = S[SA[lb]..SA[lb] + len - 1]$.

node	id	len	lb	size	suffix_lb	pred_list
CGAATC	1	6	26	3	17	[]
CAA	2	3	20	3	20	[3, 7, 9]
TCA	3	3	36	3	36	[4, 1, 1]
TTC	4	3	41	2	41	[]
CACCA	7	5	23	1	25	[3]
CAGCA	9	5	24	1	34	[3]

Figure 4: The implicit representation of the subgraph in Figure 3.

2.3 Determining node identifiers with the two bit vectors B_r and B_l

To determine whether or not a node is already stored in the subgraph, we must be able to compute its identifier efficiently. Beller and Ohlebusch [3] proposed a method with which a node identifier can be derived from two bit vectors, called B_r and B_l . The algorithm that constructs B_r and B_l in linear time can be found in [3] and will not be repeated here. Each of the two bit vectors B_r and B_l is preprocessed so that rank-queries can be answered in constant time. For our purposes, it suffices to explain the role of the bit vectors. As an example, we consider the concatenation $S = S^1 \# S^2 \# S^3$ of the three strings S^1 , S^2 , and S^3 of the preceding section. The bit vectors B_r and B_l for S can be found in Table 1. The compressed de Bruijn graph of S is shown in Figure 2. Let us start with the following observation ([3, Lemma 1]):

Let v be a node in the compressed de Bruijn graph and let ω be the string corresponding to v . If v is not the start node, then it has at least two different predecessors if and only if the length k prefix of ω is a left-maximal repeat. It has at least two different successors if and only if the length k suffix of ω is a right-maximal repeat.

To make use of this observation, the left- and right boundary of each right-maximal k -mer interval is marked by a 1 in the bit vector B_r . In our example, the 3-mers ATC,

⁵ It should be possible to construct a ‘symmetric’ subgraph by using bi-directional search [18, 25]. However, the memory requirements would be much higher because two index data structures are necessary: the wavelet tree of the BWT of S and the wavelet tree of the BWT of the reverse of S .

CAA, TCA, TTC, and TTT are right-maximal repeats. For instance, the ATC-interval is [17..19] and therefore $B_r[17] = 1$ and $B_r[19] = 1$; see Table 1. In other words, B_r has the property that $B_r[i] = 1$ if and only if the k -mer $x = S[SA[i]..SA[i] + k - 1]$ is a right-maximal repeat and $S_{SA[i]}$ is either the lexicographically smallest or the lexicographically largest suffix of S that has x as a prefix. Consequently, there are $rightMax := rank_1(B_r, n)/2$ many nodes with a right-maximal k -mer suffix in the compressed de Bruijn graph. The nodes are numbered 1 to $rightMax$ according to the lexicographic ordering of their k -mer suffixes. In our example, the node with the 3-mer suffix ATC has the number 1, etc.

Given the suffix array interval $[i..j]$ of a k -mer, Algorithm 1 tests in line 3 whether the k -mer is right-maximal or not. If so, the identifier of the corresponding node in the compressed de Bruijn graph is calculated by $rank_1(B_r, j)/2$. Now consider a node u in the compressed de Bruijn graph and the k -mer suffix of the string corresponding to u . If this k -mer is *not* right-maximal, then there must be a successor v of u that has at least two different predecessors. In this case, one can use the bit vector B_l to compute the identifier of u . B_l has the property that $B_l[i] = 1$ if and only if (a) the k -mer $x = S[SA[i]..SA[i] + k - 1]$ is *not* a right-maximal repeat, (b) $S_{SA[i]}$ is the lexicographically largest suffix of S that has x as a prefix (in other words, i is the right boundary of the x -interval in the suffix array), and (c) the k -mer $y = S[SA[i] + 1..SA[i] + k]$ is a left-maximal repeat. In our scenario, x is the k -mer suffix of u and y is the k -mer prefix of v . In the example, the 3-mer $x = ACG$ is not right-maximal and the 3-mer $y = CGA$ is a left-maximal repeat. Thus, at the right boundary of the x -interval [14..15] there is a 1 in the bit vector B_l (Algorithm 1 tests this in line 5). There are $leftMax := rank_1(B_l, n)$ many nodes of this kind in the compressed de Bruijn graph. The nodes are numbered $rightMax + 1$ to $rightMax + leftMax$ according to the lexicographic ordering of their k -mer suffixes.

Algorithm 1 shows how the identifier of a node is calculated by means of the k -mer interval $[i..j]$. We exemplify each of the three possible cases. The TCA-interval in the suffix array is [36..38]. Since $B_r[36] = 1$ and $B_r[38] = 1$, the first case applies and FIND-ID returns $id = rank_1(B_r, 38)/2 = 3$. The GCA-interval is a singleton interval at index 34 in the suffix array. So FIND-ID is called with parameters $i = j = 34$. The second case applies because $B_r[34] = 0$ and $B_l[34] = 1$. Hence FIND-ID returns the value $id = rightMax + rank_1(B_l, 34) = 5 + 4 = 9$. Finally, FIND-ID(31, 33) returns $\perp$ for the GAA-interval [31..33] because none of the two preceding cases applies. The undefined value $\perp$ indicates that there is no node that has GAA as a suffix.

Algorithm 1 Let $[i..j]$ be the suffix array interval of a k -mer substring x of S . If x is the k -mer suffix of a string corresponding to a node u in the compressed de Bruijn graph, then this algorithm, applied to $[i..j]$, returns the identifier id of u . Otherwise, it returns $\perp$.

```

1: function FIND-ID( $i, j$ )
2:    $id \leftarrow \perp$ 
3:   if  $B_r[i] = 1$  and  $B_r[j] = 1$  then
4:      $id \leftarrow rank_1(B_r, j)/2$ 
5:   else if  $B_l[j] = 1$  then
6:      $id \leftarrow rightMax + rank_1(B_l, j)$ 
7:   return  $id$ 

```

2.4 Generating a node

Let id be the identifier of a node v and let $[lb..rb]$ be the suffix array interval of the k -mer suffix of the string corresponding to v . If v is not stored in the data structure G yet (i.e., $G[id] = \perp$), then Algorithm 2 computes the components of v and stores it at $G[id]$. In lines 4 and 5, the first two components of v get initial values that will be updated later. By contrast, in lines 6 and 7, the last two components of v get initial values that will not change anymore. In the while-loop, the string ω corresponding to v is (implicitly) computed. Let ω be the string $S[SA[v.lb]..SA[v.lb] + v.len - 1]$ (initially ω is the k -mer $S[SA[lb]..SA[lb] + k - 1]$) and consider an execution of the body of the while-loop. The procedure call $getIntervals([v.lb .. v.lb + v.size - 1])$ delivers the list of all $c\omega$ -intervals, where $c \in \Sigma$. If list contains more than one element, then ω is left-maximal. In this case, $extendable = false$ and the while-loop terminates. Otherwise, if list contains just one element $(c, [i..j])$, then ω is not left-maximal. If $c = \#$ or $c = \$$, then the beginning of a genome (or chromosome) is reached and again the while-loop terminates. Otherwise, the computation must proceed. According to [3, Lemma 2], the $c\omega$ -interval $[i..j]$ is not a subinterval of a right-maximal k -mer interval if and only if $rank_1(B_r, i)$ is even and $B_r[i] = 0$. Algorithm 2 tests this in line 14. If the test returns *true*, it sets $extendable$ to *true*, increments $v.len$ by one and sets the value of $v.lb$ to i . This is correct because the next iteration the while-loop corresponds to the string $c\omega$ and the $c\omega$ -interval is $[i..i + v.size - 1]$.

Note that the components B , *colors*, and *visited* are initialized at the end of Algorithm 2. These components will be explained later.

We exemplify Algorithm 2 by computing the components of the node with identifier $id = 1$ (so we assume node 1 is not stored in the subgraph yet). The call $EXTEND(1, 17, 19, 3, G)$ uses the parameters $lb = 17$ and $rb = 19$ because [17..19] is the ATC-interval in the suf-

Algorithm 2 Extend

```

1: function EXTEND( $id, lb, rb, k, G$ )
2:   if  $G[id] = \perp$  then
3:     initialize a new node
4:      $node.len \leftarrow k$ 
5:      $node.lb \leftarrow lb$ 
6:      $node.size \leftarrow rb - lb + 1$ 
7:      $node.suffix\_lb \leftarrow lb$ 
8:      $extendable \leftarrow \text{true}$ 
9:     while  $extendable$  do
10:       $extendable \leftarrow \text{false}$ 
11:       $list \leftarrow getIntervals([node.lb .. node.lb + node.size - 1])$ 
12:      if  $list$  contains just one element  $(c, [i..j])$  and  $c \notin \{\#, \$\}$  then
13:         $ones \leftarrow rank_1(B_r, i)$ 
14:        if  $ones$  is even and  $B_r[i] = 0$  then
15:           $extendable \leftarrow \text{true}$ 
16:           $node.len \leftarrow node.len + 1$ 
17:           $node.lb \leftarrow i$ 
18:         $node.B[1..node.size] \leftarrow 0$ 
19:         $node.colors[1..node.size] \leftarrow 0$ 
20:         $node.visited \leftarrow \text{false}$ 
21:       $G[id] \leftarrow node$ 

```

fix array (see Table 1) and ATC is the 3-mer suffix of the string corresponding to node 1. Since BWT[17..19] solely contains A's, the procedure *getIntervals* returns only the AATC-interval [10..12], which is not a subinterval of a right-maximal k -mer. In the next iteration of the while-loop, *getIntervals* delivers just the GAATC-interval [31..33] because BWT[10..12] is a sequence of G's. Again, this interval is not a subinterval of a right-maximal k -mer and the while-loop is executed once more. Because BWT[31..33] contains no character other than C, *getIntervals* again returns just one interval, namely the CGAATC-interval [26..28]. Once again, this interval is not a subinterval of a right-maximal k -mer. In the next iteration of the while-loop, the current string is not extendable anymore because BWT[26..28] contains two different characters. At that point in time, we have $node.len = 6$ (the length of the string CGAATC is six) and $node.lb = 26$ (the left boundary of the CGAATC-interval).

2.5 Finding a predecessor node

Given a node $v = G[id]$, which corresponds to string ω , we must be able to find its predecessor nodes. Since the $k - 1$ length suffix of a predecessor node equals the $k - 1$ length prefix of ω , the k -mer suffix of a predecessor node is the k -mer prefix $c\omega[1..k]$ of a string $c\omega$, where $c \in \Sigma$. Given $c \in$

Σ , Algorithm 3 returns the identifier of the c -predecessor of v , say node u . This works as follows. Backward search, applied to the ω -interval $[lb..rb]$, yields the $c\omega$ -interval $[i..j]$. Note that $c\omega$ is not a k -mer because $|c\omega| > k$. Therefore, Algorithm 1 is not applicable. However, it is possible to compute the identifier with the help of [3, Lemma 2], which states that the $c\omega$ -interval $[i..j]$ is a subinterval of a right-maximal k -mer interval if and only if $rank_1(B_r, i)$ is odd or $B_r[i] = 1$.

Note that the k -mer prefix $c\omega[1..k]$ of $c\omega$ coincides with the k -mer suffix of the string corresponding to u . Clearly, the $c\omega$ -interval $[i..j]$ is a subinterval of the $c\omega[1..k]$ -interval. Let $ones = rank_1(B_r, i)$. If $c\omega[1..k]$ is right-maximal, then, by [3, Lemma 2], the identifier of u is $new_id = \lfloor (ones + 1)/2 \rfloor$, but the interval boundaries of the $c\omega[1..k]$ -interval are not known. If $c\omega[1..k]$ is not right-maximal, then ω must be left-maximal and the identifier of u is $new_id = rightMax + rank_1(B_l, i - 1) + 1$; see line 9 in Algorithm 3. In this case, the interval boundaries of the $c\omega[1..k]$ -interval are known already because the $c\omega$ -interval $[i..j]$ coincides with the $c\omega[1..k]$ -interval; see [3, Lemma 3].

If node u is not stored in G yet (i.e., $G[new_id] = \perp$), then Algorithm 3 computes all components of u and stores it in G by calling the procedure EXTEND. As already mentioned, if $c\omega[1..k]$ is right-maximal, the interval bound-

Algorithm 3 Find predecessor

```

1: function FIND-PREDECESSOR( $c, id, k, G$ )
2:    $lb \leftarrow G[id].lb$ 
3:    $rb \leftarrow lb + G[id].size - 1$ 
4:    $[i..j] \leftarrow \text{backwardSearch}(c, [lb..rb])$ 
5:    $ones \leftarrow \text{rank}_1(B_r, i)$ 
6:   if  $ones$  is odd or  $B_r[i] = 1$  then
7:      $new\_id \leftarrow \lfloor (ones + 1)/2 \rfloor$ 
8:   else
9:      $new\_id \leftarrow \text{rightMax} + \text{rank}_1(B_l, i - 1) + 1$ 
10:  if  $G[new\_id] = \perp$  then
11:    if  $ones$  is odd then
12:      if  $B_r[i] = 0$  then
13:         $i \leftarrow \text{select}_1(B_r, ones)$ 
14:         $j \leftarrow \text{select}_1(B_r, ones + 1)$ 
15:      else if  $B_r[i] = 1$  then  $\triangleright ones$  is even
16:         $i \leftarrow \text{select}_1(B_r, ones - 1)$ 
17:  EXTEND( $new\_id, i, j, k, G$ )
18:  return  $new\_id$ 

```

aries of the $c\omega[1..k]$ -interval must be calculated before **EXTEND** is called. This can be done by a case analysis:

1. If $ones$ is odd and $B_r[i] = 0$, then $B_r[i..j]$ solely contains zeros. Thus, the left boundary of the $c\omega[1..k]$ -interval is $\text{select}_1(B_r, ones)$ and the right boundary is $\text{select}_1(B_r, ones + 1)$.
2. If $ones$ is odd and $B_r[i] = 1$, then the left boundary of the $c\omega[1..k]$ -interval is i and the right boundary is $\text{select}_1(B_r, ones + 1)$.
3. If $ones$ is even and $B_r[i] = 1$, then the left boundary of the $c\omega[1..k]$ -interval is $\text{select}_1(B_r, ones - 1)$ and the right boundary is i . It should be stressed that in this case $i = j$ must hold.

Then, the procedure call **EXTEND**(new_id, i, j, k, G) completes the job.

We illustrate Algorithm 3 with the help of the running example. The node with $id = 2$ in Figure 3 has three predecessors. We are looking for the G -predecessor of node 2. This is the node whose corresponding string ends with the 3-mer GCA. One backwards search step with the character G and the CAA-interval [20..22] yields the GCAA-interval [34..34]. We then determine the identifier new_id of the G -predecessor of node 2, which is 9. Finally, the procedure call **EXTEND**(9, 34, 34, 3, G) first calculates the components of node 9 and then stores it into the subgraph.

2.6 Finding a path

If the user specifies a starting position *start* and an end position *end* in S , then the corresponding *reference path* in the compressed de Bruijn graph can be determined by Algorithm 4. Before we explain the algorithm, we introduce additional components of a node, which are very useful for displaying the subgraph. Let $v = G[id]$ be a node in the subgraph and let ω be the string corresponding to v . The new components are:

- *pred_list*, the modified set of the predecessors of v . If u is the c -predecessor of v , then we store $\langle u.id, c, [i..j] \rangle$ in $v.pred_list$, where $[i..j]$ is the $c\omega$ -interval.
- B , a bit vector of size $v.size$ indicating which positions correspond to a path that is continued (the last edge of a path will be visualized differently, so that the user can easily see where a path ends). Recall that ω occurs at each position in the set $\{SA[q] \mid v.lb \leq q \leq v.lb + v.size - 1\}$. Fix an index q with $v.lb \leq q \leq v.lb + v.size - 1$ and let $p = SA[q]$. The length $k - 1$ suffix of the occurrence of ω starting at position p in S can be found at position $p' = p + v.len - (k - 1)$. If there is a successor v' of v in the subgraph so that $p' = SA[q']$ for some q' with $v'.lb \leq q' \leq v'.lb + v'.size - 1$, then we say that the path that includes position p can be continued with position p' (if ω' is the string corresponding to v' , then ω and ω' occur consecutively in one of the genomes with an overlap of $k - 1$ characters). Note that v is the c -predecessor of v' , where $c = S[p' - 1]$. Thus, $\langle v.id, c, [i..j] \rangle$ is in $v'.pred_list$, where $[i..j]$ is the $c\omega'$ -interval. Note that $[i..j]$ is a subinterval of $[v.suffix_lb..v.suffix_lb + v.size - 1]$ and that $a = \text{LF}(p')$ is in $[i..j]$. The ‘relative’ position in the interval $[v.suffix_lb..v.suffix_lb + v.size - 1]$ corresponding to the ‘absolute’ position a (in the interval $[1..n]$) is $r = a - v.suffix_lb + 1$, and we set $B[r]$ to one. The key observation is that $q = v.lb + r - 1$. So we have indirectly marked the index q , which represents the position p because $p = SA[q]$.
- *colors*, an array of size $v.size$, which is defined by $v.color[r] = t$ if and only if $SA[r + v.lb - 1]$ belongs to the t -th genome. (Each genome corresponds to a color and in the visualization of the subgraph the edges will be colored accordingly.)
- *visited*, a flag that is set to *true* after the node has been visited.

The components *pred_list* and B are modified during the execution of Algorithm 4, whereas the components *color*

and *visited* are only used in the final construction of the subgraph (Algorithm 5).

Now we are in position to explain Algorithm 4. First, the suffix array interval $[i..j]$ of the k -mer suffix $y = S[end - k + 1..end]$ of $S[start..end]$ is determined by a backward search. Second, the algorithm searches for the *id* of the node whose corresponding string ω contains y . It is important to note that only the k -mer prefix of ω can be left-maximal and only the k -mer suffix of ω can be right-maximal (otherwise, ω would not be the string of a node in the de Bruijn graph because it would be split). Thus, if y is a suffix of ω , then $\text{FIND-ID}(i, j)$ (Algorithm 1) returns the *id*; otherwise it returns $\perp$. If y is not a suffix of ω (i.e., if $\text{FIND-ID}(i, j)$ returns $\perp$), then there is a suffix u of ω that has y as prefix. The key observation is that $[i..j]$ is the suffix array interval of u . Moreover, u can be written as $c_1 c_2 \dots c_\ell x$, where $\ell \geq 1$, $c_q \in \Sigma$ for $q \in \{1, \dots, \ell\}$, and x is the k -mer suffix of u . Note that the value of ℓ is unknown. Since $c_2 \dots c_\ell x$ and its k -mer prefix are not left-maximal, it follows that $[\Psi(i)..\Psi(j)]$ is the suffix array interval of both; cf. [3]. If $\ell \geq 2$, then $\text{FIND-ID}(\Psi(i), \Psi(j))$ will return $\perp$. Algorithm 4 iterates this process until the x -interval $[\Psi^\ell(i)..\Psi^\ell(j)]$ is encountered, where $\Psi^\ell(i)$ denotes the ℓ -fold application of Ψ to i , and $\text{FIND-ID}(\Psi^\ell(i), \Psi^\ell(j))$ returns the identifier of the node that corresponds to ω .

Third, $\text{EXTEND}(id, i, j, k, G)$ generates the node $G[id]$ and stores it into the subgraph. With the *getIntervals* procedure, the algorithm searches for all predecessors of $v = G[id]$. Consider a c -predecessor node u of v . In line 19, Algorithm 4 checks whether c occurs at the current position in the region $S[start..end]$. If this is the case, it continues the search with u . Otherwise, it pushes u onto a stack to process it later. The second while-loop stops when the beginning of the region $S[start..end]$ is encountered. At that point in time, the subgraph contains all nodes of the reference path (and their predecessor lists specify the reference path). Furthermore, all the predecessor nodes of the nodes in the reference path are stored in the stack. In lines 29 to 38, the algorithm processes the nodes on the stack similar to the nodes of the reference path. It determines predecessors and pushes them onto the stack, albeit with a decremented depth parameter. In this way, it generates the depth d subgraph.

Let us illustrate the algorithm by means of our example (in which $d = 2$). We are interested in the region $S[7..12] = S^1[7..12] = \text{CACCAA}$ (i.e. $start = 7$ and $end = 12$), which does neither contain the separator symbol $\#$ nor the sentinel $\$$. The 3-mer suffix of the region of interest is CAA. A backward search determines the CAA-interval $[20..22]$. From a previous example, we know that $\text{FIND-ID}(22, 24)$ returns the identifier $id = 2$. Algorithm 2 gener-

ates and stores node 2 into the subgraph. Moreover, $ref.id$ is set to id . The search for the reference path then continues with the next position $ref.pos = end - k = 12 - 3 = 9$. Since $ref.pos > start$, the body of the second while-loop is executed. The procedure *getIntervals* returns a list that contains the CCAA-interval $[25..25]$, the GCAA-interval $[34..34]$, and the TCAA-interval $[36..36]$. Calls to FIND-PREDECESSOR return the corresponding node identifiers 7, 9, and 3, respectively. Because $S[ref.pos] = S[9] = C$, node 7 is the next node in the reference path, the other nodes are pushed onto the stack. Thus, $\langle 7, C, [25..25] \rangle$ is added to the predecessor list of node 2. Moreover, $B[1]$ is set to 1 in node 7 to indicate that this (relative) position is continued (there is an edge $7 \rightarrow 2$ corresponding to this position). Before the next iteration of the second while-loop, we have $ref.id = 7$ and $ref.pos = 6$, therefore it terminates. In the third while-loop, the neighborhood of the reference path is calculated. If Algorithm 4 was called with $start = 7$, $end = 12$, and $d = 2$, it will discover the nodes 1, 3, 4, and 9 as illustrated in Figure 3.

2.7 Displaying a subgraph

Algorithm 5 displays the compressed de Bruijn subgraph G . It starts with the last node in the reference path, called *end_node*, and puts it into an initially empty queue. In the while-loop, as long as the queue is not empty, a node v is dequeued and processed. The predecessor nodes in $v.pred_list$ are taken into account on line 12 and they will eventually enter the queue. Since a node may be encountered more than once, it must be avoided that the very same node is reported several times. For this reason, every node has a flag *visited* that is set after processing the node. A node enters the queue only if its flag is not set.

When a node v is processed, all its incoming edges are reported. In the final display, an edge should be colored, so that the user can immediately see to which genome an edge belongs. For simplicity, we assume that there are m colors, numbered 1 through m . An edge $u \rightarrow v$ that belongs to genome S^t gets the color t . Let $p_1, \dots, p_m$ be the positions in S , so that the string S^t starts at position p_t in S . In the following, it will be useful to define $p_{m+1} = n$. We will say that a position p in S belongs to S^t if $p_t \leq p < p_{t+1}$. Similarly, we will say that an index i in the suffix array belongs to S^t if $p_t \leq SA[i] < p_{t+1}$. We could store this information in an array D (the so-called document array) by setting $D[i] = t$. However, such an array takes $n \log m$ bits. Since only small parts of it are required to construct the colored subgraph, we will take a different approach. Recall that every node v has an array

Algorithm 4 Given the order k , the depth $d > 0$ and the start and end positions $start$ and end of the reference region $S[start..end]$, find the path ref corresponding to $S[start..end]$ in the compressed de Bruijn graph. We assume that the string $S[start..end]$ does not contain a separator symbol.

```

1: function FIND-REFERENCE-PATH( $start, end, k, d, G$ )
2:    $[i..j] \leftarrow \text{backwardSearch}(S[end - k + 1..end], [1..n])$   $\triangleright k$ -mer suffix of  $S[start..end]$ 
3:    $id \leftarrow \text{FIND-ID}(i, j)$ 
4:    $\ell \leftarrow 0$ 
5:   while  $id = \perp$  do  $\triangleright$  search for the node that contains the  $k$ -mer suffix
6:      $i \leftarrow \Psi(i)$   $\triangleright \Psi$  is the inverse of LF
7:      $j \leftarrow \Psi(j)$ 
8:      $\ell \leftarrow \ell + 1$ 
9:      $id \leftarrow \text{FIND-ID}(i, j)$ 
10:   $\text{EXTEND}(id, i, j, k, G)$   $\triangleright$  now  $G[id]$  is known
11:   $ref.id \leftarrow id$ 
12:   $ref.len \leftarrow G[id].len - \ell$ 
13:   $ref.pos \leftarrow end - ref.len$ 
14:  initialize empty stack
15:  while  $ref.pos \geq start$  do
16:     $list \leftarrow \text{getIntervals}([G[ref.id].lb .. G[ref.id].lb + G[ref.id].size - 1])$ 
17:    for each  $(c, [i..j])$  with  $c \notin \{\#, \$\}$  in  $list$  do
18:       $id \leftarrow \text{FIND-PREDECESSOR}(c, ref.id, k, G)$ 
19:      if  $c = S[ref.pos]$  then
20:         $ref.len \leftarrow ref.len + G[id].len - (k - 1)$ 
21:         $tmp.pos \leftarrow end - ref.len$ 
22:         $tmp.id \leftarrow id$ 
23:      else
24:         $push(\langle id, d - 1 \rangle)$ 
25:         $G[ref.id].pred\_list.append(\langle id, c, [i..j] \rangle)$ 
26:         $G[id].B[i - G[id].suffix\_lb + 1 .. j - G[id].suffix\_lb + 1] \leftarrow 1$   $\triangleright$  all entries are set to 1
27:       $ref.id \leftarrow tmp.id$ 
28:       $ref.pos \leftarrow tmp.pos$ 
29:   $push(\langle ref.id, d - 1 \rangle)$ 
30:  while stack is not empty do
31:     $\langle id, d \rangle \leftarrow pop()$ 
32:    if  $d > 0$  then
33:       $list \leftarrow \text{getIntervals}([G[id].lb .. G[id].lb + G[id].size - 1])$ 
34:      for each  $(c, [i..j])$  with  $c \notin \{\#, \$\}$  in  $list$  do
35:         $pred\_id \leftarrow \text{FIND-PREDECESSOR}(c, id, k, G)$ 
36:         $push(\langle pred\_id, d - 1 \rangle)$ 
37:         $G[id].pred\_list.append(\langle pred\_id, c, [i..j] \rangle)$ 
38:         $G[pred\_id].B[i - G[pred\_id].suffix\_lb + 1 .. j - G[pred\_id].suffix\_lb + 1] \leftarrow 1$ 

```

$v.colors$ of size $v.size$ as a component, which is defined by $v.colors[p] = t$ if and only if $SA[p + v.lb - 1]$ belongs to S^t (note that p is now a relative position, the corresponding absolute position is $p + v.lb - 1$). Clearly, $v.colors[p]$ can be computed in $O(\log m)$ time by a binary search of $SA[p + v.lb - 1]$ within the sorted array $[p_1, \dots, p_{m+1}]$. Fortunately, many of the binary searches can be avoided by

propagating color-information to the predecessor nodes. To do this efficiently, we employ an array $idx_map[1..\sigma]$ of lists. For each index $q \in \{1, \dots, v.size\}$ we determine the character $c = \text{BWT}[q + v.lb - 1]$ and store q in the list $idx_map[c]$. That is, the list $idx_map[c]$ contains all (relative) positions that correspond to the c -predecessor node u of v . The identifier of u is the first component of

Algorithm 5 Display subgraph

```

1: function REPORT-SUBGRAPH( $G, k, \text{end\_node}, S, \text{SA}, \text{BWT}$ )
2:    $\text{sorted\_array} \leftarrow \text{sort}(1, \text{SA}[1], \text{SA}[2] + 1, \text{SA}[3] + 1, \dots, \text{SA}[m] + 1)$   $\triangleright \text{SA}[1] = n$ 
3:   initialize an empty queue
4:    $\text{enqueue}(\text{end\_node})$ 
5:   while queue is not empty do
6:      $\text{node} \leftarrow \text{dequeue}()$ 
7:     if  $\text{node.visited} \neq \text{true}$  then
8:       report  $\text{node.id}$  and  $S[\text{SA}[\text{node.lb}].. \text{SA}[\text{node.lb}] + \text{node.len} - 1]$   $\triangleright$  report  $id$  and  $text$ 
9:       initialize  $\text{idx\_map}[1..\sigma]$  as an array of empty lists
10:      for  $q \leftarrow 1$  to  $\text{node.size}$  do
11:         $\text{idx\_map}[\text{BWT}[q + \text{node.lb} - 1]].\text{append}(q)$ 
12:      for each  $\langle \text{pid}, c, [i..j] \rangle$  in  $\text{node.pred\_list}$  do
13:        for  $\ell \leftarrow 1$  to  $\text{idx\_map}[c].\text{size}$  do
14:           $p \leftarrow \text{idx\_map}[c][\ell]$ 
15:          report  $\text{pid} \rightarrow \text{node.id}$   $\triangleright$  report edge
16:          if  $\text{node.B}[p] = 0$  then
17:            report path end
18:          if  $\text{node.colors}[p] = \perp$  then
19:             $\text{node.colors}[p] = \text{BINARY-SEARCH}(\text{SA}[p + \text{node.lb} - 1], \text{sorted\_array})$ 
20:             $G[\text{pid}].\text{colors}[i + \ell - G[\text{pid}].\text{suffix\_lb}] \leftarrow \text{node.colors}[p]$ 
21:            report  $\text{node.colors}[p]$ 
22:          if  $G[\text{pid}].\text{visited} = \text{false}$  then
23:             $\text{enqueue}(G[\text{pid}])$ 
24:           $G[\text{pid}].\text{visited} \leftarrow \text{true}$ 

```

the triple $\langle \text{pid}, c, [i..j] \rangle$ in $v.\text{pred_list}$ and the interval $[i..j]$ is a subinterval of $[u.\text{suffix_lb}..u.\text{suffix_lb} + u.\text{size} - 1]$. Moreover, if p is the ℓ -th element in the list $\text{idx_map}[c]$, then $\text{LF}[p + v.\text{lb} - 1] = i + \ell - 1$. That is, $p + v.\text{lb} - 1$ and $i + \ell - 1$ belong to the same genome. In other words, $u.\text{colors}[i + \ell - u.\text{suffix_lb}] = v.\text{colors}[p]$. Consequently, the edge $u \rightarrow v$, which is reported in line 15, gets the color $v.\text{colors}[p]$ in line 21.

In our opinion, it is also important to display the end of a path. Here the bit vector $v.B$ comes into play. If $v.B[p] = 1$, then there is an edge corresponding to position $p + v.\text{lb} - 1$. Hence the path will be continued. Otherwise, the path ends and we use a dot as arrowhead to illustrate this in the graph.

Let us illustrate Algorithm 5 with the compressed de Bruijn subgraph G from our example. The sorted_array contains the elements $[1, 14, 27, 45]$ and the end node v of the reference path has the identifier $id = 2$. This node v is the first that is processed in the while-loop. Clearly, v was not visited yet. In line 8, Algorithm 5 reports the node identifier 2 and the string CAA corresponding to node v . To propagate color-information to the predecessor nodes of node v , we need the array $\text{idx_map}[1..\sigma]$. The CAA-interval is

$[20..22]$ and $\text{BWT}[20..22] = \text{TCG}$. Therefore, $\text{idx_map}[\text{T}] = [1], \text{idx_map}[\text{C}] = [2]$, and $\text{idx_map}[\text{G}] = [3]$. Let us confine ourselves to the first predecessor node of v . The corresponding triple in the predecessor list $v.\text{pred_list}$ is $\langle 3, \text{T}, [36..36] \rangle$. The algorithm reports the edge $3 \rightarrow 2$ and, since $v.B[1] = 0$, it also reports that the path ends here. The next step is to find the color of the edge. Since $v.\text{colors}[1] = \perp$, a binary search determines the index t in the sorted_array that satisfies $\text{sorted_array}[t] \leq \text{SA}[20] = 42 < \text{sorted_array}[t + 1]$. It returns $t = 3$, thus the algorithm sets $v.\text{colors}[1] = 3$. With the help of idx_map , it is possible to infer $u.\text{colors}[36 - u.\text{suffix_lb} + \text{idx_map}[\text{T}][1]] = u.\text{colors}[1] = v.\text{colors}[1] = 3$, where u is the node with identifier 3. In line 21, Algorithm 5 reports 3 as the color of the edge $3 \rightarrow 2$ (when the graph is displayed, the arrow in the graph is colored accordingly).

Figure 5: Compressed de Bruijn subgraph of a region of the *toIA* gene of some *E. coli* strains. The components of the node 69934 :: GGTAT..(+562)..GCGCG are its identifier 69934, the first and last five nucleotides of the DNA-sequence and the number (+562) of nucleotides in between. To improve readability, the starting positions of the DNA-sequence in the *E. coli* genomes are not displayed, but they are delivered by the program.

3 Results

Our implementation is based on the `sdsl-lite` library [11]. The program is divided into two subprograms. The first subprogram reads an input file in FASTA format, constructs the suffix array, the BWT, and the wavelet tree of the BWT (including support of rank- and select-queries in constant time), and stores them on secondary memory. The second subprogram reads the wavelet tree from disk and then proceeds by a case distinction: if, for the specified value of k , the bit-vectors B_r and B_l (including support of rank- and select-queries) are known already, then it loads them from disk; otherwise it computes them. After that, the subprogram constructs (and displays) the subgraph according to the parameters specified by the user (start and end position of the region for which the reference path shall be built and a value d for the d -neighborhood). The second subprogram keeps only the wavelet tree of the BWT and the bit-vectors B_r and B_l in main memory.

Two experiments were conducted on a 64 bit Ubuntu 14.04.1 LTS (Kernel 3.13) system equipped with two 16-core Intel(R) Xeon(TM) E5-2698v3 CPU with 2.8 GHz and 256 GB of RAM (but no parallelism was used). The programs were compiled with g++ (version 8.1.0) using the provided makefile. Our software and test data are available at <https://www.uni-ulm.de/in/theo/research/seqana.html>.

In the first experiment, we used four *E. coli* genomes (accession numbers FM180568, CP000247, CP001671, and CP000468). The parameters were $k = 100$, $d = 5$, $start = 799800$ and $end = 800300$. The region of interest is part of the highly variable *tolA* gene in the *E. coli* strains; see [28]. The generated subgraph can be found in Figure 5. To evaluate the performance of our software, we repeated the test 50 times. In all cases, the memory peak did not exceed 45 MB of RAM usage. On average, the initialization of the index structures took 7.8 seconds, the construction of the subgraph required only 1.6 seconds.

In the second experiment, we used 10 human genomes: (i) five different assemblies of the human reference genome (UCSC Genome Browser assembly IDs: hg16, hg17, hg18, hg19, and hg38), (ii) the maternal and paternal haplotype of individual NA12878 (Utah female) of the 1000 Genomes Project [24], and (iii) three long read (PacBio) assemblies specified at our website. The size of the 10 human genomes was 29 GB. For the construction of the subgraph, we used the parameters $k = 50$, $d = 3$, $start = 13907712200$ and $end = 13907714200$. The region under consideration is part of the large repetitive central exon of the human MUC5AC gene; see [13]. We repeated this test 5 times and report average timings. The

construction of the index with the semi-external algorithm of [4] took about 8.5 hours and the memory peak during the computation was 38.5 GB (and the index of size 198 GB was stored on secondary memory). The construction of the subgraph (including the computation of the bit-vectors B_r and B_l) was finished in 24.5 minutes and required 39.6 GB of main memory; the subgraph itself required merely 15 KB of memory. A picture of the subgraph can be found at our website. If one wants to generate a subgraph for a different value of k , it is not necessary to rebuild the index (which takes hours), but it suffices to recompute the bit-vectors B_r and B_l , which requires about 10 minutes.

4 Discussion

The Computational Pan-Genomics Consortium [27] strongly recommended that a pan-genome data structure should support the following functionality: “All information within a data structure should be easily accessible for human eyes by visualization support on different scales.” If a pan-genome graph has many nodes (from tens of thousands to millions), such an amount of information is certainly not easily accessible for human eyes. Thus, we developed a technique to construct pan-genome subgraphs on the fly. The demand for visualization of pan-genomes on different scales can be met in our approach: if one wants to build compressed de Bruijn subgraphs for different values of k , it is not necessary to rebuild the index structure. One merely has to compute two bit-vectors (on which the construction of the subgraph is based) for the desired value of k and the subgraph G containing the reference path and its d -neighborhood. The time to build the bit-vectors and G is much less than the time needed to construct the index structure. In summary, in our approach it is possible to efficiently represent and visualize pan-genome subgraphs at different granularities.

Acknowledgement: We thank the anonymous reviewers and Thomas Büchler for their helpful and constructive comments. Furthermore, we are grateful to Uwe Baier for his help with the `sdsl`. We would also like to thank Ben Langmead for pointing us to the long read assemblies and the DFG (OH 53/7-1) for funding.

References

- [1] U. Baier, T. Beller, and E. Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler

- transform. *Bioinformatics*, 32(4):497–504, 2016.
- [2] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.
 - [3] T. Beller and E. Ohlebusch. A representation of a compressed de Bruijn graph for pan-genome analysis that enables search. *Algorithms for Molecular Biology*, 11:20, 2016.
 - [4] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. Space-efficient construction of the Burrows-Wheeler transform. In *Proc. 20th International Symposium on String Processing and Information Retrieval*, volume 8214 of *Lecture Notes in Computer Science*, pages 5–16. Springer, 2013.
 - [5] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, 1994.
 - [6] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
 - [7] A. Dilthey, C. Cox, Z. Iqbal, M.R. Nelson, and G. McVean. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47(6):682–688, 2015.
 - [8] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *Lecture Notes in Computer Science*, pages 697–710. Springer, 2010.
 - [9] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.
 - [10] T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.
 - [11] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014.
 - [12] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.
 - [13] X. Guo, S. Zheng, H. Dang, R.G. Pace, J.R. Stonebraker, C.D. Jones, F. Boellmann, G. Yuan, P. Haridass, O. Fedrigo, D.L. Corcoran, M.A. Seibold, S.S. Ranade, M.R. Knowles, W.K. O’Neal, and J.A. Voynow. Genome reference and sequence variation in the large repetitive central exon of human MUC5AC. *American Journal of Respiratory Cell and Molecular Biology*, 50(1):223–232, 2014.
 - [14] G. Holley, R. Wittler, and J. Stoye. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11:3, 2016.
 - [15] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, 2012.
 - [16] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.
 - [17] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
 - [18] T.-W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional BWT. In *Proc. International Conference on Bioinformatics and Biomedicine*, pages 31–36. IEEE Computer Society, 2009.
 - [19] S. Marcus, H. Lee, and M.C. Schatz. SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.
 - [20] M.D. Muggli, A. Bowe, N.R. Noyes, P.S. Morley, K.E. Belk, R. Raymond, T. Gagie, S.J. Puglisi, and C. Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.
 - [21] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
 - [22] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009.
 - [23] S.J. Puglisi, W.F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.
 - [24] J. Rozowsky, A. Abyzov, J. Wang, P. Alves, D. Raha, A. Harmanci, J. Leng, R. Bjornson, Y. Kong, N. Kitabayashi, N. Bhardwaj, M. Rubin, M. Snyder, and M. Gerstein. AlleleSeq: Analysis of allele-specific expression and binding in a network framework. *Molecular Systems Biology*, 7:522, 2011.
 - [25] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proc. 21st Annual Symposium on Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 40–50. Springer, 2010.
 - [26] H. Tettelin, V. Maignani, M.J. Cieslewicz, C. Donati, D. Medini, N.L. Ward, S.V. Angiuoli, J. Crabtree, A.L. Jones, A.S. Durkin, et al. Genome analysis of multiple pathogenic isolates of *Streptococcus agalactiae*: implications for the microbial pan-genome. *Proceedings of the National Academy of Sciences of the United States of America*, 102(39):13950–13955, 2005.
 - [27] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2016.
 - [28] K. Zhou, K. Vanoirbeek, A. Aertsen, and C.W. Michiels. Variability of the tandem repeat region of *Escherichia coli* *tolA* gene. *Research In Microbiology*, 163:316–322, 2012.