Bioinformatics, 32(4), 2016, 497–504
doi: 10.1093/bioinformatics/btv603
Advance Access Publication Date: 26 October 2015
Original Paper

OXFORD

Genome analysis

Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform

Uwe Baier, Timo Beller and Enno Ohlebusch*

Institute of Theoretical Computer Science, Ulm University, 89069 Ulm, Germany

*To whom correspondence should be addressed.

Associate Editor: John Hancock

Received on July 30, 2015; revised on September 17, 2015; accepted on October 13, 2015

Abstract

Motivation: Low-cost genome sequencing gives unprecedented complete information about the genetic structure of populations, and a population graph captures the variations between many individuals of a population. Recently, Marcus *et al.* proposed to use a compressed de Bruijn graph for representing an entire population of genomes. They devised an $O(n \log g)$ time algorithm called splitMEM that constructs this graph directly (i.e. without using the uncompressed de Bruijn graph) based on a suffix tree, where n is the total length of the genomes and g is the length of the longest genome. Since the applicability of their algorithm is limited to rather small datasets, there is a strong need for space-efficient construction algorithms.

Results: We present two algorithms that outperform splitMEM in theory and in practice. The first implements a novel linear-time suffix tree algorithm by means of a compressed suffix tree. The second algorithm uses the Burrows–Wheeler transform to build the compressed de Bruijn graph in $O(n \log \sigma)$ time, where σ is the size of the alphabet. To demonstrate the scalability of the algorithms, we applied it to seven human genomes.

Availability and implementation: https://www.uni-ulm.de/in/theo/research/seqana/.

Contact: Enno.Ohlebusch@uni-ulm.de

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

1.1 Background

Second- and third-generation sequencers produce vast amounts of DNA sequence information, and it is often the case that multiple genomes of the same or closely related species are available. An example is the 1000 Genomes Project, which started in 2008. Its goal was to sequence the genomes of at least 1000 humans from all over the world and to produce a catalog of all variations (SNPs, indels, etc.) in the human population. In this article, the term 'pan-genome' of the population refers to the genomic sequences together with this catalog. Tettelin *et al.* (2005) coined the term pan-genome a decade ago; they evaluated the composition of six strains of *Streptococcus agalactiae*. The pan-genome analysis of other bacteria followed: e.g. the pan-genome structure of *Escherichia coli* was studied by Rasko *et al.* (2008). In a broader sense, the pan-genome defines the entire genomic repertoire of a given phylogenetic clade (which may range from species to phylum and beyond). 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.

Since the *de novo* assembly of, e.g. mammalian genomes, is still a serious problem (both from a technological and a budgetary point of view), the reference-based approach dominates in genomics. Small wonder that most methods to represent a pan-genome by a graph are reference based. Here, we briefly discuss a few of them: Schneeberger *et al.* (2009) were the first to explicitly model variation in a population DAG of a few *Arabidopsis thaliana* genomes.

Rahn *et al.* (2014) used a data structure called 'Journaled String Tree' to consistently represent both SNPs and indels as edits to a reference genome, and Dilthey *et al.* (2015) use a population reference graph for genome inference. In contrast to these works, which are all alignment-based, Paten *et al.* (2014) came up with a different solution: they proposed context mapping to relate genomes.

Marcus *et al.* (2014) proposed a reference- and alignment-free approach for pan-genome analyses. Ideally, it takes multiple *assembled* genomes as input, but it can also work with *contigs*. For some species, especially medically important bacteria, multiple complete genomes are available. For example, in 2015, NCBI GenBank contained 72 strains of *Chlamydia trachomatis* (a sexually transmitted human pathogen) and 62 strains of the prokaryotic model organism *E.coli*. Marcus *et al.* (2014) proposed a compressed de Bruijn graph as a graphical representation of the relationship between genomes. Basically, it is a compressed version of the colored de Bruijn graph introduced by Iqbal *et al.* (2012). Marcus *et al.* (2014) describe an $O(n \log g)$ time algorithm that directly computes the compressed de Bruijn graph based on a suffix tree (ST), where n is the total length of the genomes and g is the length of the longest genome. They write about their software splitMEM: 'Future work remains to improve splitMEM and further unify the family of sequence indices. Although . . . , most desired are techniques to reduce the space consumption . . . ' In this article, we present two different techniques that achieve this goal. The first implements a novel linear-time suffix tree algorithm by means of a compressed ST (CST). The second algorithm uses the Burrows–Wheeler transform to build the compressed de Bruijn graph in $O(n \log \sigma)$ time, where σ is the size of the alphabet Σ. Preliminary ideas for the second algorithm were presented by Beller and Ohlebusch (2015). In contrast to splitMEM, both of our algorithms use only $O(n)$ space.

The *contracted* de Bruijn graph introduced by Cazaux *et al.* (2014) is not identical with the *compressed* de Bruijn graph. A node in the contracted de Bruijn graph is not necessarily a substring of one of the genomic sequences [see the remark following Definition 3 in the article by Cazaux *et al.* (2014)]. Thus the contracted de Bruijn graph, which can be constructed in linear time from the ST, is not useful for our purposes.

Very recently, other alignment-free and reference-free approaches were described by Solomon and Kingsford (2015) and Holley *et al.* (2015). Both are k-mer approaches that use Bloom filters. We will discuss their relationship to the splitMEM approach in Section 4.

1.2 Problem definition

Given a string S of length n and a natural number k, the de Bruijn graph representation of S contains a node for each distinct length k substring of S, called a k-mer. Two nodes u and v are connected by a directed edge $u \to v$ if u and v occur consecutively in S, i.e. $u = S[i..i+k-1]$ and $v = S[i+1..i+k]$, where $S[i..j]$ denotes the substring of S starting with the character at position i and ending with the character at position j. Figure 1 shows an example. Clearly, the graph contains at most n nodes and n edges. By construction, adjacent nodes will overlap by $k-1$ characters, and the graph can include multiple edges connecting the same pair of nodes or self-loops representing overlapping repeats. For every node, except for the start node (containing the first k characters of S) and the stop node (containing the last k characters of S), the in-degree coincides with the out-degree. A de Bruijn graph can be 'compressed' by merging non-branching chains of nodes into a single node 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 $u \to v$), 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 node) has at least two different predecessors or its single predecessor has at least two different successors and every node (apart from the stop node) has at least two different successors or its single successor has at least two different predecessors; Figure 1. 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 we will build it directly.

Figure 2 shows how splitMEM represents the compressed de Bruijn graph G for $k=3$ and the string S =ACTACGTACGTACG$. 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 (sorted in such a way that the walk through G that gives S is induced by the adjacency lists: if node $G[id]$ is visited for the ith time, then its successor is the node that can be found at position i in the adjacency list of $G[id]$).

In pan-genome analysis, S is the concatenation of multiple genomic sequences, where the different sequences are separated by special symbols (in practice, we use one separator symbol and treat the different occurrences of it as if they were different characters). The nodes in the compressed de Bruijn graph of a pan-genome can be categorized as follows:

- A uniqueNode represents a unique substring in the pan-genome and has a single start position (i.e. *posList* contains just one element).
- A repeatNode represents a substring that occurs at least twice in the pan-genome, either as a repeat in a single genome or as a segment shared by multiple genomes.

According to Marcus *et al.* (2014), the compressed de Bruijn graph is most suitable for pan-genome analysis: 'This way the complete pan-genome will be represented in a compact graphical representation such that the shared/strain-specific status of any substring is immediately identifiable, along with the context of the flanking sequences. This strategy also enables powerful topological analysis of the pan-genome not possible from a linear representation'. Figure 3 illustrates this point of view.

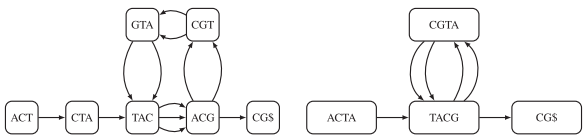

Fig. 1. The de Bruijn graph for $k=3$ and the string ACTACGTACGTACG$ is shown on the left, while its compressed counterpart is shown on the right

node	*id*	*len*	*posList*	*adjList*
CGTA	1	4	[5, 9]	[2, 2]
TACG	2	4	[3, 7, 11]	[1, 1, 4]
ACTA	3	4	[1]	[2]
CG$	4	3	[13]	[ ]

Fig. 2. Representation of the compressed de Bruijn graph from Fig. 1

1.3 Suffix trees and other index data structures

In this section, we briefly introduce the data structures on which our new algorithms are based. For details, we refer to the textbooks of Gusfield (1997) and Ohlebusch (2013), and the references therein.

An ST for a string S of length n is a compact trie storing all the suffixes of S, i.e. the concatenation of the edge labels on the path from the root to leaf i exactly spells out the ith suffix $S_i = S[i..n]$ of S (S is terminated with the special character \$ to guarantee that each suffix ends at a leaf of the tree). An ST can be built in linear time. For each node v in ST, $str(v)$ denotes the string obtained by concatenating the labels on the edges on the root-to-v path. In this article, we need the following operations on STs (u and v are nodes): $sDepth(v)$ gives v's string-depth $|str(v)|$; $parent(v)/fChild(v)/nSibling(v)$ yields the parent/first child/next sibling of v (if existent); if $str(v) = a\alpha$ for some character a, then the suffix link $sLink(v)$ gives the unique node u with $str(u) = \alpha$; $\texttt{LCA}(u, v)$ yields the lowest common ancestor of u and v; $\texttt{HAQ}(v, k)$ returns the highest ancestor u of v so that $|str(u)| \geq k$. Below we will show that $\texttt{HAQ}(v, k)$ can be supported in constant time; it is well-known that the same is true for the other operations.

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, i.e. it satisfies $S_{\texttt{SA}[1]} < S_{\texttt{SA}[2]} < \cdots < S_{\texttt{SA}[n]}$; Figure 4. A suffix array can be constructed in linear time. For every substring ω of S, the ω-interval is the suffix array interval $[i..j]$ so that ω is a prefix of $S_{\texttt{SA}[k]}$ if and only if $i \leq k \leq j$. For a node v in ST, the $str(v)$-interval in the suffix array, denoted by $[lb(v)..rb(v)]$, contains all the suffixes of the subtree of ST rooted at v.

To support the operation $\texttt{HAQ}(v, k)$ on the ST in constant time, initialize a bit vector B of size n with zeros and proceed as follows. For each internal node u in ST with $sDepth(u) \geq k$ and $sDepth(parent(u)) < k$ set $B[lb(u)] = 1$ and $B[rb(u)] = 1$, where $[lb(u)..rb(u)]$ is the $str(u)$-interval. Now preprocess B in linear time so that a $rank_1(B, i)$ query (returns the number of ones in B up to and including position i) and a $select_1(B, i)$ query (returns the position of the ith one in B) can be answered in constant time. The resulting data structure requires only $n + o(n)$ bits: n bits for B and $o(n)$ bits to support the queries in constant time. The node $u = \texttt{HAQ}(v, k)$ can then be found in constant time as follows: $select_1(rank_1(B, lb(v)))$ returns the position i of the first 1 in B that is left to $lb(v)$ and $select_1(rank_1(B, lb(v)) + 1)$ returns the position j of the first 1 in B that is right to $rb(v)$ (note that $i = lb(v)$ and $j = rb(v)$ is possible). It can be shown that leaf i is the leftmost and leaf j is the rightmost leaf of the subtree rooted at u. Hence $\texttt{LCA}(i, j)$ yields the node u (note that $u = v$ is possible).

The Burrows–Wheeler transform converts the string S into the string $\texttt{BWT}[1..n]$ defined by $\texttt{BWT}[i] = S[\texttt{SA}[i] - 1]$ for all i with $\texttt{SA}[i] \neq 1$ and $\texttt{BWT}[i] =$ \$ otherwise; Figure 4. The BWT can be built in linear time via the suffix array, but there are also algorithms that construct the BWT directly (i.e. without constructing the suffix array).

The LF mapping (last-to-first-mapping) is defined as follows: If $\texttt{SA}[i] = q$, then $\texttt{LF}(i)$ is the index j so that $\texttt{SA}[j] = q - 1$ (if $\texttt{SA}[i] = 1$, then $\texttt{LF}(i) = 1$). In other words, if the ith entry in the suffix array is the suffix S_q, then $\texttt{LF}(i)$ 'points' to the entry at which the suffix S_{q-1} can be found; Figure 4. In data compression, the LF-mapping is used to reconstruct the original string S from the BWT (given the BWT, the LF-mapping can easily be computed in linear time).

The LCP-array stores the lengths of the *longest common prefixes* of lexicographically adjacent suffixes: for $2 \leq i \leq n$, $\texttt{LCP}[i] = \max\{p \geq 0 | S_{\texttt{SA}[i]}$ and $S_{\texttt{SA}[i-1]}$ share a prefix of length $p\}$ and $\texttt{LCP}[1] = -1 = \texttt{LCP}[n + 1]$; see Figure 4 for an example. The LCP-array can be computed in linear time from the suffix array and its inverse, but it is also possible to construct it directly from the BWT in $O(n \log \sigma)$ time.

A CST with full functionality supports the same operations as an ST. It consists of three components: a compressed suffix array, a compressed LCP-array and a succinct representation of the ST topology. A CST requires much less space than an ST, but it cannot support all operations in constant time.

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 $|\{\texttt{BWT}[q] | i \leq q \leq j\}| \geq 2$, i.e. the set $\{S[\texttt{SA}[q] - 1] | i \leq q \leq j\}$ of all characters that precede at least one of

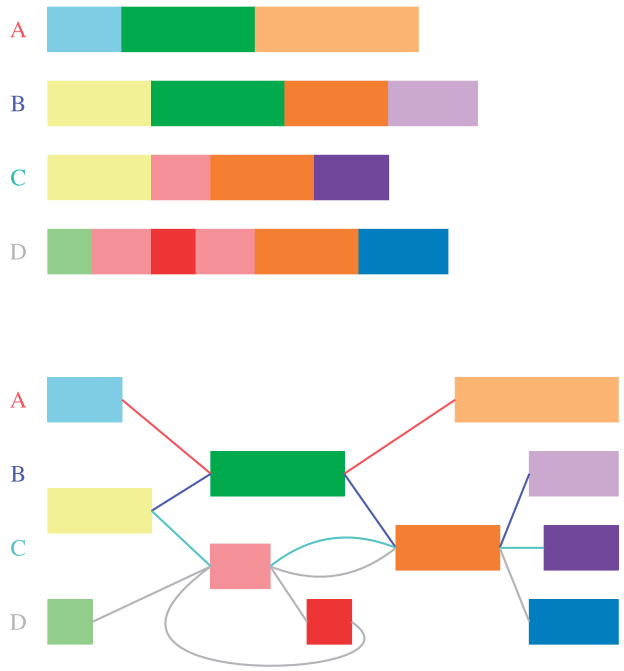

Fig. 3. Cartoon representation of a pan-genome consisting of the four genomes A–D. The genomic sequences are decomposed into strain-specific segments and segments that are shared by a subset of the strains. The edges maintain the adjacencies of the segments

i	SA	LCP	B_1	B_2	B_3	LF	BWT	$S_{\texttt{SA}[i]}$
1	15	-1	0	0	0	10	G	\$
2	12	0	1	0	0	13	T	ACG\$
3	8	3	0	0	0	14	T	ACGTACG\$
4	4	7	1	0	0	15	T	ACGTACGTACG\$
5	1	2	0	0	0	1	\$	ACTACGTACGTACG\$
6	13	0	0	0	0	2	A	CG\$
7	9	2	0	0	0	3	A	CGTACG\$
8	5	6	0	0	0	4	A	CGTACGTACG\$
9	2	1	0	0	1	5	A	CTACGTACGTACG\$
10	14	0	0	0	0	6	C	G\$
11	10	1	0	0	0	7	C	GTACG\$
12	6	5	0	0	1	8	C	GTACGTACG\$
13	11	0	0	1	0	11	G	TACG\$
14	7	4	0	1	0	12	G	TACGTACG\$
15	3	8	0	1	0	9	C	TACGTACGTACG\$
16		-1						

Fig. 4. The suffix array SA of the string ACTACGTACGTACG\$ and related notions are defined in Section 1.3. The bit vectors B_1, B_2 and B_3 are explained in Section 2.2

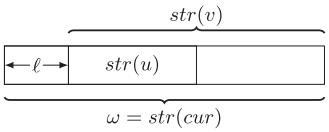

Fig. 5. Suppose that the first ℓ characters of the repeat $\omega = str(cur)$ have already been considered in the repeat-loop of Algorithm 1. Then $str(v) = \omega[\ell+1..|\omega|]$ and $str(u)$ is the shortest prefix of $str(v)$ of length $\geq k$ that is right-maximal because $u = \text{HAQ}(v,k)$

the suffixes $S_{\text{SA}[i]}, \ldots, S_{\text{SA}[j]}$ is not singleton (where $S[0] := \$$). Analogously, the repeat ω is *right-maximal* if $|\{S[\text{SA}[q]+\ell] | i \leq q \leq j\}| \geq 2$. A left- and right-maximal repeat is called *maximal* repeat. [Note that Marcus *et al.* (2014) use the term 'maximal exact match' instead of the more common term 'maximal repeat'. We will not use the term 'maximal exact match' here.] An internal node v in the ST is called left-maximal if $str(v)$ is a left-maximal repeat. Note that every internal node v of the ST is right-maximal in the sense that $str(v)$ is a right-maximal repeat.

2 Methods

Our algorithms are based on the following simple lemma:

> *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.*

The lemma provides a tool to decide at which positions a split occurs. Both of our methods make extensive use of it.

2.1 Using a (compressed) suffix tree

Marcus *et al.* (2014) describe an algorithm that computes the compressed de Bruijn graph directly (i.e. without using the uncompressed de Bruijn graph) based on an ST of the pan-genome. Their algorithm consists of two phases:

1. Compute the set of repeatNodes of the compressed de Bruijn graph.
2. Compute the uniqueNodes as well as edges between nodes in the compressed de Bruijn graph.

They introduce so-called 'suffix skips' to compute repeatNodes in $O(n \log g)$ time. Our new linear-time Algorithm 1 computes repeatNodes without them. The idea behind Algorithm 1 is as follows: Start the computation only with left-maximal internal nodes that have no internal node as child and use suffix links to detect their repeat-structure (i.e. where splits are necessary). Stop the computation if a left-maximal internal node x is encountered. This is because either x has no internal node as child, so it will be considered later, or x has an internal node y as child, so its repeat-structure will be found when y is processed. In Algorithm 1, *cur* is a pointer to a repeat that must or must not be split later, and ℓ is the number of characters that must be skipped before the split occurs (note that if a split occurs, then the left and right part overlap by $k-1$ characters). A split occurs if the k-length prefix of $str(v)$ is left- or right-maximal, where v is the current internal node. Let $u = \text{HAQ}(v,k)$ and consider the situation of Figure 5. In that situation, Algorithm 1 uses the following case analysis:

Algorithm 1. Computation of the repeatNodes in the compressed de Bruijn graph based on an ST

```
1: function COMPUTE-REPEAT-NODES(k,ST)
2:     mark each left-maximal internal node of the ST
3:     for each internal node v that is left-maximal and
       whose children are all leaves do
4:         d ← sDepth(v)
5:         if d = k then
6:             CREATE-REP-NODE(v,k)
7:         else if d > k then
8:             cur ← v
9:             ℓ ← 0
10:            repeat
11:                u ← HAQ(v, k)              ▷ u = v is possible
12:                if u is left-maximal then
13:                    if ℓ > 0 then                    ▷ Case 1
14:                        CREATE-REP-NODE (cur, ℓ+k−1)
15:                    if sDepth(u) = k then            ▷ Case 1a
16:                        CREATE-REP-NODE (u, k)
17:                        cur ← sLink(v)
18:                        ℓ ← 0
19:                    else          ▷ Case 1b: sDepth(u) ≠ k
20:                        cur ← v
21:                        ℓ ← 1
22:                else                  ▷ u is not left-maximal
23:                    if sDepth(u) = k then            ▷ Case 2a
24:                        CREATE-REP-NODE (cur, ℓ+k)
25:                        cur ← sLink(v)
26:                        ℓ ← 0
27:                    else          ▷ Case 2b: sDepth(u) ≠ k
28:                        ℓ ← ℓ+1
29:                v ← sLink(v)
30:                d ← d − 1
31:            until v is left-maximal or d = k
32:            if ℓ > 0 then
33:                if v is left-maximal then
34:                    CREATE-REP-NODE (cur, ℓ+k−1)
35:                else
36:                    CREATE-REP-NODE (cur, ℓ+k)
37:            else                                  ▷ ℓ = 0
38:                if v is not left-maximal then
39:                    CREATE-REP-NODE (v, k)
```

1 If $str(u)$ is a left-maximal repeat, then the length k prefix of $str(u)$ is left-maximal. This implies that the length k prefix of $str(v)$ is left-maximal. If in this case $\ell > 0$, then ω must be split at the beginning of $str(u)$; so the length $\ell + k - 1$ prefix of ω is a repeatNode.

1a If $str(u)$ is a right-maximal k-mer (because $sDepth(u) = k$), then ω must also be split at the end of $str(u)$ because $str(u)$ is also a repeatNode. The algorithm continues with $\omega[\ell+2..|\omega|]$ and $\ell = 0$.

1b If $str(u)$ is not a right-maximal k-mer, then the algorithm continues with the string $str(v) = \omega[\ell+1..|\omega|]$ and $\ell = 1$.

2a If $str(u)$ is not a left-maximal repeat but a right-maximal k-mer, then ω must be split at the end of $str(u)$; so the length $\ell + k$ prefix of ω is a repeatNode. The algorithm continues with $\omega[\ell+2..|\omega|]$ and $\ell = 0$.

2b If $str(u)$ is neither a left-maximal repeat nor a right-maximal k-mer, then the algorithm continues with the string ω and $\ell + 1$.

The procedure CREATE-REP-NODE in Algorithm 1 creates a new repeatNode if this node does not exist yet.

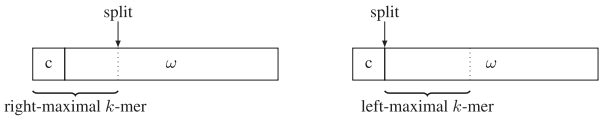

Fig. 6. The string ω is prefix of the suffix S_p of S and $c\omega$ is prefix of S_{p-1}. ω must be split if (i) the length k prefix of $c\omega$ is a right-maximal repeat or (ii) the length k prefix of ω is a left-maximal repeat

We will next show that Algorithm 1 runs in $O(n)$ time. Since all operations in the algorithm take constant time, the run-time is proportional to the overall number of suffix links that are followed. The ST of a string S of length n has n leaves. Since every internal node is branching, ST has at most $n-1$ internal nodes. It follows that there are at most $n-1$ suffix links because every internal node has exactly one suffix link. We claim that every suffix link is used at most once in Algorithm 1. Suppose to the contrary that a suffix link from node v to node w is used more than once. This is only possible if node v has at least two incoming suffix links, say from nodes u and u'. If $str(v) = \alpha$, then we must have $str(u) = a\alpha$ and $str(u') = a'\alpha$ for two distinct characters a and a'. However, this implies that v is left-maximal and Algorithm 1 stops whenever a left-maximal node is reached. This contradiction proves our claim and shows that Algorithm 1 runs in linear time.

Once Algorithm 1 has computed the repeatNodes, we proceed as in the second phase of splitMEM: the set of genomic starting positions that occur in each repeatNode is sorted, so that uniqueNodes that bridge any gaps between adjacent repeatNodes as well as the edges in the compressed de Bruijn graph can be computed in a single pass over the sorted list. However, there is one difference to Marcus *et al.* (2014): to achieve linear run-time, we use a non-comparison-based sorting algorithm for this task; details can be found in the Supplementary Material.

2.2 Using the BWT

Our second algorithm uses the BWT and the LF-mapping to compute the complete compressed de Bruijn graph G (uniqueNodes, repeatNodes and the edges between them) in a single backward pass over the whole pan-genome S. To be more precise, Algorithm 2 starts with the suffix $S_n = \$$ at index $j = 1$ in the suffix array and successively computes the indices of $S_{n-1}, \ldots, S_1$ with the help of the LF mapping (i.e. S_{n-1} can be found at index $i = \mathtt{LF}[j]$, S_{n-2} can be found at index $\mathtt{LF}[i]$, etc.). In Algorithm 2, the current string ω is a prefix of suffix S_p, which occurs at index j in the suffix array. The next string that must be considered in the algorithm is $c\omega$, where $c = S[p-1]$. Note that $c\omega$ is a prefix of suffix S_{p-1}, which occurs at index $i = \mathtt{LF}(j)$ in the suffix array. The string ω must be split if (i) the length k prefix of $c\omega$ is a right-maximal repeat or (ii) the length k prefix of ω is a left-maximal repeat; see Figure 6 for an illustration.

Algorithm 2 uses three bit vectors B_1, B_2 and B_3, which are constructed in a preprocessing phase by the procedure CREATE-BIT-VECTORS. Here, we only briefly explain this procedure; it is described in detail in the Supplementary Material, where it is also shown that its run-time is $O(n \log \sigma)$. All three bit vectors are initialized with zeros. The procedure obtains the bit vector B_1 by computing the suffix array interval $[lb..rb]$ of each right-maximal k-mer and setting $B_1[lb] = B_1[rb] = 1$; see Figure 4 for an example. Moreover, B_1 is preprocessed, so that rank-queries can be answered in constant time. By means of B_1, it is then possible to perform test (i) in constant time: if $c\omega$ is prefix of the suffix at index i in SA, then it has a right-maximal k-mer as prefix if and only if $B_1[i] = 1$ or $rank_1(B_1, i)$ is odd. If this is the case, a split occurs and Algorithm 2 must continue with the k-mer prefix of $c\omega$ as next node. The number $\lfloor (rank_1(B_1, i) + 1)/2 \rfloor$ will serve as a unique identifier of this next node. In the following, $rightMax = rank_1(B_1, n)/2$ is the number of right-maximal k-mer intervals. Procedure CREATE-BIT-VECTORS also computes the bit vectors B_2 and B_3 as follows: If the suffix array interval $[lb..rb]$ of a left-maximal repeat of length $\geq k$ is detected (hence the corresponding k-mer prefix is left-maximal), then $B_2[q]$ is set to 1 for all q in $[lb..rb]$. Moreover, for each c in $\mathtt{BWT}[lb..rb]$, the procedure sets $B_3[\mathtt{LF}[q]]$ to 1, where q is the index of the last occurrence of c in $\mathtt{BWT}[lb..rb]$. Finally, the procedure resets each one bit in B_3 that marks a right-maximal k-mer to zero (the reason for this will become clear in a moment); in the example of Figure 4, no bit of B_3 had to be reset. By means of B_2, it is then possible to perform test (ii) in constant time: if ω is prefix of the suffix at index j in SA, then it has a left-maximal k-mer as prefix if and only if $B_2[j] = 1$. If this is the case, a split occurs and Algorithm 2 must continue with the k-mer prefix x of $c\omega$, which is a prefix of the suffix at index $i = \mathtt{LF}[j]$, as next node. If $B_1[i] = 1$ or $rank_1(B_1, i)$ is odd, then $\lfloor (rank_1(B_1, i) + 1)/2 \rfloor$ is the identifier of this next node. If not, then we use the bit vector B_3 to assign the unique identifier $rightMax + rank_1(B_3, i-1) + 1$ to the next node, which corresponds to (or ends with) x. This is because $rightMax$ is the number of all right-maximal k-mers and $rank_1(B_3, i-1) = rank_1(B_3, lb'-1)$, where $[lb'..rb']$ is the x-interval. It can be shown that $B_3[lb'..rb'-1]$ solely contains zeros and $B_3[rb'] = 1$; consequently $rank_1(B_3, i-1) + 1 = rank_1(B_3, rb')$.

To sum up, after the preprocessing phase it is known that the compressed de Bruijn graph G has $rightMax + leftMax + 1$ many nodes: there are $rightMax = rank_1(B_1, n)/2$ many nodes that end with a right-maximal k-mer, $leftMax = rank_1(B_3, n)$ many nodes that end with a non-right-maximal k-mer and the stop node that ends with the special symbol \$. Consequently, Algorithm 2 initializes an array G of that size, in which a node is represented by the triple $(len, posList, adjList)$, where $posList$ is the sorted list of positions at which the corresponding string ω occurs in S, len is the length of ω and $adjList$ is the corresponding adjacency list. The for-loop of Algorithm 2 implements the single backward pass over S as described above. A split occurs whenever $number \neq \perp$. In this case, the position p is added to the front of the $posList$ of the current node cur and cur is added to the front of the $adjList$ of the next node $number$. If $number = \perp$, then the length of the string corresponding to node cur is incremented by one.

Algorithm 2. Construction of a compressed de Bruijn graph without the suffix array.

```
1: function CREATE-COMPRESSED-GRAPH(k, BWT, LF)
2:     (B_1, B_2, B_3) ← CREATE-BIT-VECTORS (k, BWT)
3:     rightMax ← rank_1(B_1, n)/2
4:     leftMax ← rank_1(B_3, n)
5:     create an array G of size rightMax + leftMax + 1
6:     j ← 1                          ▷ suffix $ occurs at index 1
7:     cur ← rightMax + leftMax + 1   ▷ id of the stop node
8:     G[cur].len ← 1                 ▷ length of the suffix $
9:     for p ← n down to 2 do
10:        i ← LF(j)                  ▷ LF is the last-to-first mapping
11:        ones ← rank_1(B_1, i)
12:        number ← ⊥
13:        if ones is odd or B_1[i] = 1 then
14:            number ← ⌊(ones + 1)/2⌋
15:        else if B_2[j] = 1 then
16:            number ← rightMax + rank_1(B_3, i − 1) + 1
17:        if number ≠ ⊥ then
18:            add p to the front of G[cur].posList
19:            add cur to the front of G[number].adjList
20:            G[number].len ← k
21:            cur ← number
22:        else
23:            increment G[cur].len by one
24:        j ← i
25:    add 1 to the front of G[cur].posList
```

As explained in the Supplementary Material, the computation of the bit vectors B_1, B_2 and B_3 requires $O(n \log \sigma)$ time. Apart from the LF-mapping, all operations in Algorithm 2 take only constant time. In our implementation, the LF-mapping is implemented by a wavelet tree of the BWT, so it takes $O(\log \sigma)$ time to compute a value LF(j). Consequently, the overall run-time of Algorithm 2 is $O(n \log \sigma)$.

2.3 The size of the compressed de Bruijn graph

It follows from the preceding section that the size of the compressed de Bruijn graph can be characterized in terms of left- and right-maximal k-mer repeats. The number of nodes equals $|V_1| + |V_2| + 1$, where $V_1 = \{\omega | \omega$ is a right-maximal k-mer repeat in $S\}$ and $V_2 = \{\omega | \exists i \in \{1, \ldots, n-k\} : \omega = S[i..i+k-1] \notin V_1$ and $S[i+1..i+k]$ is a left-maximal k-mer repeat in $S\}$; the stop node is taken into account by adding 1. The number of edges is $|\{i | 1 \leq i \leq n-k$ and $S[i..i+k-1] \in V_1 \cup V_2\}|$.

3 Results

We implemented our new algorithms in C++, using the library sdsl of Gog *et al.* (2014). Software and test data are available at http://www.uni-ulm.de/in/theo/research/seqana.html. Both algorithms use a variant of the semi-external algorithm described in Beller *et al.* (2013b) to construct the CST and the BWT, respectively. The experiments were conducted on a 64 bit Ubuntu 14.04.1 LTS (Kernel 3.13) system equipped with two ten-core Intel Xeon processors E5-2680v2 with 2.8 GHz and 128 GB of RAM (but no parallelism was used). All programs were compiled with g++ (version 4.8.2) using the provided makefile.

With the CST-based and the BWT-based algorithm, respectively, we built compressed de Bruijn graphs for the 62 *E.coli* genomes (containing 310 million base pairs) listed in the Supplementary Material of Marcus *et al.* (2014), using the k-mer lengths 50, 100 and 1000. Table 1 lists the results of our experiments. The run-times include the construction of the index, but similar to splitMEM it is unnecessary to rebuild the index for a fixed dataset and varying values of k. The peak memory usage reported in Table 1 includes the size of the index *and* the size of the compressed de Bruijn graph. Because of its large memory requirements, splitMEM was not able to build a compressed de Bruijn graph for all 62 strains of *E.coli* on our machine equipped with 128 GB of RAM. That is why we included a comparison based on the first 40 *E.coli* genomes (containing 199 million base pairs) of the dataset. The experimental results show that both of our algorithms use significantly less space (two orders of magnitude) than splitMEM. The CST-based algorithm is five times faster than splitMEM, while the BWT-based algorithm is more than an order of magnitude faster. It is worth mentioning that our two algorithms compute isomorphic—but not necessarily identical—compressed de Bruijn graphs because the node identifiers may differ.

To show the scalability of our new algorithms, we applied them to five different assemblies of the human reference genome (UCSC Genome Browser assembly IDs: hg16, hg17, hg18, hg19 and hg38) as well as the maternal and paternal haplotype of individual NA12878 (Utah female) of the 1000 Genomes Project; see Rozowsky *et al.* (2011). The compressed de Bruijn graphs of their first chromosomes (7xChr1, containing 1736 million base pairs) and the complete seven genomes (7xHG, containing 21 201 million base pairs) were built for the k-mer lengths 50, 100 and 1000. The experimental results in Table 1 show that the BWT-based algorithm clearly outperforms the CST-based algorithm. It took slightly over 6 h (22 000 s) to construct the index of the seven human genomes and less than 2 h (6000–7000 s) to build the graph with the BWT-based algorithm for these genomes and a specific value of k (50, 100 or 1000). Table 2 lists some statistics about the compressed de Bruijn graphs.

In pan-genome analysis, $S = S^1 \# S^2 \# \ldots S^{m-1} \# S^m \$$ is the concatenation of multiple genomic sequences $S^1, \ldots, S^m$, separated by a special symbol #. (In theory, one could use pairwise different symbols to separate the sequences, but in practice this would blow up the alphabet.) This has the effect that # may be part of a repeat.

Table 1. Construction of compressed de Bruijn graphs

	$k = 50$	$k = 100$	$k = 1000$
40 *E.coli*			
splitMEM	1985 (572.19)	2098 (572.20)	1653 (572.19)
CST-based	473 (4.91)	448 (4.72)	401 (4.55)
BWT-based	185 (2.22)	184 (1.63)	194 (1.49)
62 *E.coli*			
splitMEM	—	—	–
CST-based	755 (4.57)	721 (4.42)	641 (4.09)
BWT-based	331 (2.26)	310 (1.68)	329 (1.49)
7xChr1			
splitMEM	—	—	—
CST-based	4488 (4.50)	4501 (4.46)	4296 (4.44)
BWT-based	1776 (3.08)	1748 (2.75)	1734 (2.62)
7xHG			
splitMEM	—	—	—
CST-based	87 605 (4.74)	82 812 (4.62)	80 116 (4.58)
BWT-based	29 014 (2.78)	28 129 (2.22)	28 588 (2.05)

The columns show the run-times in seconds and, in parentheses, the maximum main memory usage in bytes per base pair. A minus indicates that the algorithm was not able to solve its task on our machine equipped with 128 GB of RAM.

Table 2. Statistics about the compressed de Bruijn graphs

	$k = 50$	$k = 100$	$k = 1000$
40 *E.coli*			
graph size	1.07	0.65	0.06
edges	9 205 701	5 157 748	301 191
nodes	767 391	552 240	79 252
uniqueNodes	129 901	104 951	26 322
repeatNodes	637 490	447 289	52 930
avg. out-degree	12.00	9.34	3.80
avg. node length	89.67	175.42	2232.84
avg. uNode length	146.87	276.05	3299.91
avg. rNode length	78.018	151.81	1702.19
62 *E.coli*			
graph size	1.12	0.68	0.06
edges	16 304 084	9 219 061	555 810
nodes	1 007 765	738 980	117 021
uniqueNodes	174 717	141 167	34 463
repeatNodes	833 048	597 813	82 558
avg. out-degree	16.18	12.48	4.75
avg. node length	86.70	170.15	2105.87
avg. uNode length	132.23	257.81	3242.76
avg. rNode length	77.15	149.45	1631.28
7xChr1			
graph size	1.87	1.59	1.50
edges	193 620 506	167 393 470	160 874 818
nodes	1 718 646	939 054	310 841
uniqueNodes	215 190	195 241	91 221
repeatNodes	1 503 456	743 813	219 620
avg. out-degree	112.66	178.26	517.55
avg. node length	186.16	371.53	2310.18
avg. uNode length	104.25	212.23	2500.46
avg. rNode length	197.88	413.35	2231.14
nodes shared by 1	14.76%	24.15%	29.44%
nodes shared by 2	6.89%	11.03%	12.51%
nodes shared by 3	0.18%	0.28%	0.40%
nodes shared by 4	0.49%	0.64%	0.88%
nodes shared by 5	7.86%	12.39%	17.00%
nodes shared by 6	11.23%	17.54%	20.37%
nodes shared by 7	58.60%	33.96%	19.39%
7xHG			
graph size	1.65	1.16	1.00
edges	2 056 675 301	1 475 958 859	1 319 219 774
nodes	25 367 105	12 030 826	3 851 688
uniqueNodes	2 614 834	2 316 797	1 143 848
repeatNodes	22 752 271	9 714 029	2 707 840
avg. out-degree	81.08	122.68	342.50
avg. node length	163.48	364.16	2326.95
avg. uNode length	99.44	208.65	2505.46
avg. rNode length	170.84	401.24	2251.54
nodes shared by 1	11.36%	21.23%	30.01%
nodes shared by 2	5.93%	10.74%	12.17%
nodes shared by 3	0.19%	0.31%	0.43%
nodes shared by 4	0.31%	0.47%	0.66%
nodes shared by 5	6.20%	11.60%	16.63%
nodes shared by 6	9.74%	17.46%	20.64%
nodes shared by 7	66.28%	38.19%	19.46%

The first row in a block specifies the experiment. The second row shows the graph size in bytes per base pair. Rows 3–6 contain the numbers of edges, nodes, uniqueNodes and repeatNodes, respectively. Rows 7–10 show the average out-degree of the nodes as well as the average string length of the nodes, uniqueNodes and repeatNodes. The remaining rows (if applicable) contain the percentage of the nodes that are shared by x sequences.

Our CST-based algorithm mimics splitMEM in this respect, whereas the BWT-based algorithm treats the different occurrences of # as if they were different characters. Assuming that # is the second smallest character, this can be achieved as follows. As explained in the Supplementary Material, all right-maximal k-mers can be determined without the entire LCP-array if the algorithm in Beller *et al.* (2013a) is used. If there are $m - 1$ occurrences of # in total and this algorithm starts with $m - 1$ singleton intervals $[i..i]$, $2 \leq i \leq m$, instead of the #-interval $[2..m]$, then the different occurrences of # are treated as if they were different characters.

4 Discussion

We have presented two space-efficient methods to build the compressed de Bruijn graph from scratch. An experimental comparison with splitMEM showed that our algorithms are more than an order of magnitude faster than splitMEM while using significantly less space (two orders of magnitude). To demonstrate their scalability, we successfully applied them to seven complete human genomes. Consequently, it is now possible to use the compressed de Bruijn graph for much larger pan-genomes than before (consisting, e.g. of hundreds or even thousands of different strains of bacteria). Although the BWT-based algorithm is the clear winner of the comparison, CST-based algorithms are still important. This is because STs play a central role in sequence analysis and most bioinformatics curricula comprise courses that cover this important data structure. It is therefore conceivable that a bioinformatician might be able to come up with a suffix tree algorithm that solves his/her problem at hand, but not with an algorithm that is based on the BWT and/or related data structures. If the space requirement of the ST is the bottleneck in the application, one can use a CST instead. CSTs with full functionality are, e.g. implemented in the succinct data structure library (sdsl) of Gog *et al.* (2014). On the downside, extra features such as suffix skips are not implemented in those libraries so that a direct implementation of a suffix tree algorithm by means of a CST might not be possible.

Future work includes parallel implementations of the algorithms. Moreover, it should be worthwhile to investigate the time-space trade-off if one uses data structures that are optimized for highly repetitive texts, see Navarro and Ordóñez (2014) and the references therein. Most important, however, is to address the problem of compressing the 'compressed de Bruijn graph' itself. (Our experiments show that for smaller k, the size of the graph can be larger than the size of the index, e.g. the graph for the seven human genomes and $k = 50$ takes 1.65 bytes per base pair, whereas the BWT-index requires only 1.13 bytes per base pair.) Very recently, two Bloom filter methods were presented that can be used for this purpose. Solomon and Kingsford (2015) introduced the Sequence Bloom Tree to support sequence-based querying of large-scale collections of thousands of short-read sequencing experiments and applied it to the problem of finding conditions under which query transcripts are expressed. The second approach by Holley *et al.* (2015) is closer to the splitMEM approach. Their data structure—the Bloom Filter Trie (BFT)—allows to efficiently store and traverse the *uncompressed* de Bruijn graph. In the Section Conclusion of their article, Holley *et al.* (2015) write 'Future work concerns the possibility to compress non-branching paths . . . ' This is exactly what splitMEM and our new algorithms do, so maybe the combination of both approaches will yield the ideal pan-genome representation.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. Special thanks go to Gonzalo Navarro, who initiated this work.

Funding

This work was supported by the Deutsche Forschungsgemeinschaft (DFG grant no. OH 53/6-2).

Conflict of Interest: none declared.

References

Beller,T. and Ohlebusch,E. (2015) Efficient construction of a compressed de Bruijn graph for pan-genome analysis. In: Cicalese,F. *et al*. (eds), *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching,* volume 9133 of *Lecture Notes in Computer Science*. Springer, Cham, Heidelberg, New York, Dordrecht London, pp. 40–51.

Beller,T. *et al*. (2013a) Computing the longest common prefix array based on the Burrows-Wheeler transform. *J. Discrete Algorithms*, **18**, 22–31.

Beller,T. *et al*. (2013b) Space-efficient construction of the Burrows-Wheeler transform. In: Kurland,O. *et al*. (eds) *Proceedings of the 20th International Symposium on String Processing and Information Retrieval,* volume 8214 of *Lecture Notes in Computer Science*. Springer, Cham, Heidelberg, New York, Dordrecht London, pp. 5–16.

Cazaux,B. *et al*. (2014) From indexing data structures to de Bruijn graphs. In: Kulikov,A.S. *et al*. (eds), *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching,* volume 8486 of *Lecture Notes in Computer Science*. Springer, Cham, Heidelberg, New York, Dordrecht, London, pp. 89–99.

Dilthey,A. *et al*. (2015) Improved genome inference in the MHC using a population reference graph. *Nat. Genet.*, **47**, 682–688.

Gog,S. *et al*. (2014) From theory to practice: plug and play with succinct data structures. In: Gudmundsson,J. *et al*. (eds), *Proceedings of the 13th International Symposium on Experimental Algorithms,* volume 8504 of *Lecture Notes in Computer Science*. Springer, Cham, Heidelberg, New York, Dordrecht, London, pp. 326–337.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge University Press, New York, NY, USA.

Holley,G. *et al*. (2015) Bloom filter trie—a data structure for pan-genome storage. In: Pop,M. and Touzet,H. (eds), *Proceedings of the 15th International Workshop on Algorithms in Bioinformatics,* volume 9289 of *Lecture Notes in Bioinformatics*. Springer, Heidelberg, New York, Dordrecht, London, pp. 217–230.

Iqbal,Z. *et al*. (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, **44**, 226–232.

Marcus,S. *et al*. (2014) SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, **30**, 3476–3483.

Navarro,G. and Ordóñez,A. (2014) Faster compressed suffix trees for repetitive text collections. In: Gudmundsson,J. and Katajainen,J. (eds), *Proceedings of the 13th International Symposium on Experimental Algorithms,* volume 8504 of *Lecture Notes in Computer Science*. Springer, Cham, Heidelberg, New York, Dordrecht, London, pp. 424–435.

Ohlebusch,E. (2013) *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, Bremen, Germany.

Paten,B. *et al*. (2014) Mapping to a reference genome structure, http://arxiv.org/pdf/1404.5010.pdf.

Rahn,R. *et al*. (2014) Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, **30**, 3499–3505.

Rasko,D.A. *et al*. (2008) The pangenome structure of *Escherichia coli*: comparative genomic analysis of *E. coli* commensal and pathogenic isolates. *J. Bacteriol.*, **190**, 6881–6893.

Rozowsky,J. *et al*. (2011) AlleleSeq: analysis of allele-specific expression and binding in a network framework. *Mol. Syst. Biol.*, **7**, 522.

Schneeberger,K. *et al*. (2009) Simultaneous alignment of short reads against multiple genomes. *Genome Biol.*, **10**, R98.

Solomon,B. and Kingsford,C. (2015) Large-scale search of transcriptomic read sets with sequence Bloom trees. *bioRxiv*, 017087, http://biorxiv.org/content/biorxiv/early/2015/03/26/017087.full.pdf.

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**, 13950–13955.