Pan-Genomes and de Bruijn Graphs
Seminar Report

Sebastian Gieße

Karlsruhe Institute of Technology

Abstract. This report is based on the paper "Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform" (Baier et al. 2016). The pan-genome of a population is a collection of genomic sequences of individuals in this population as well as genetic variations. Marcus et al. (2014) proposed the compressed de Bruijn graph as a suitable datastructure for the pan-genome and introduced the splitMEM algorithm to construct this graph. Baier et al. (2016) improved the splitMEM algorithm and developed two algorithms that outperformed splitMEM significantly. Ilia Minkin et al. (2016) devised a scalable, low-memory algorithm, called TwoPaCo, that was even more efficient.

1 Introduction

With the advances made in genome sequencing in the last ten years more and more genome sequences are available. Not only is the number of sequenced species growing but also the number of sequenced individuals of the same species. In October 2015 the 1000 Genomes Project contained a catalog of 2504 human genomes (Consortium et al. 2015). Finding similarities and variations in individuals of the same population is of great interest in the field of comparative genomics. A catalog of sequences and variations of individuals of the same population is called the pan-genome of a population.

Multiple Sequence Alignement (MSA) is a common method of comparing multiple sequences. A MSA consists of mutiple sequences which are aligned by insertion of gaps in order to maximize the similarity of the sequences. While MSAs can be used to identify small-scale mutations (substitution, insertion, deletion) they fail to recognize larger scale mutations like chromosomal translocations, inversions or gene duplications.

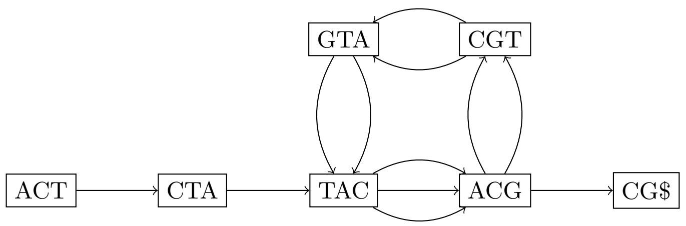
Various approaches based on de Bruijn graphs were proposed to overcome the shortcomings of MSAs (Raphael et al. 2004; Ilya Minkin et al. 2013) . Marcus et al. (2014) proposed the colored and compressed de Bruijn graph as a suitable representation for the pan-genome as "the complete pan-genome will be 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". They also developed the splitMEM algorithm to construct the compressed de Bruijn graph. Baier et al. (2016) improved the splitMEM algorithm and developed a linear time algorithm. They also developed another algorithm (bwt-based) that uses the Burrows-Wheeler transformation which was even more efficient. Using bwt-based the compressed de Bruijn graph for seven humans can be calculated in several hours. Ilia Minkin et al. (2016) introduced TwoPaCo, a scalable, low-memory algorithm that not only outperforms bwt-based by a considerable factor but also adapts to the amount of available memory.

2 Preliminaries

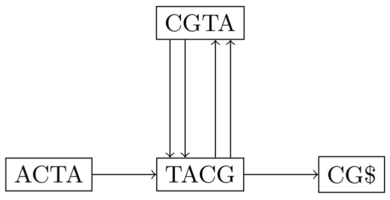
2.1 De Bruijn Graphs

Given a string s of length n we denote the substring from positions i to j (inclusive) with $s[i..j]$. A substring of length k is called a k-mer. A substring t of s with length k and $t = s[1..k]$ is a prefix of s. A substring t of s with length k and $t = s[n-k+1..n]$ is a suffix of s.

The de Bruijn graph $G(S, k)$ for a string S and a natural number k is a graph constructed as follows: For each distinct k-mer of S the de Bruijn graph contains a node representing this k-mer. If $u = S[i..i+k]$ and $v = S[i+1..i+1+k]$ u and v are connected by an edge $u \rightarrow v$. Two nodes u and v can be merged into a single node if u is the only predecessor of v and v the only successor of u. The resulting graph is a compressed de Bruijn graph.



(a)



(b)

	id	len	posList	adjList	
ACTA	1	4	{1}	{2}	uniqueNode
TACG	2	4	{3,7,11}	{3,3,4}	repeatNode
CGTA	3	4	{5,9}	{2,2}	repeatNode
CG$	4	3	{13}	{}	uniqueNode

(c)

Fig. 1: $S = ACTACGTACGTACG\$$ and $k = 3$. (a) The de Bruijn graph. (b) The compressed de Bruijn graph and (c) its representation as a list of tuples $(id, len, posList, adjList)$.

In order to compute the de Bruijn graph representation of the pan-genome for genomic sequences $S_1, S_2, \dots, S_n$ n seperator symbols $\#_1, \#_2, \dots, \#_n$ are used to determine the input string $S = S_1\#_1S_2\#_2 \dots S_n\#_n\$$. In practice these seperator symbols are often the same character to prevent the alphabet size from growing too big.

The compressed de Bruijn graph will be stored as a list of nodes. Each node corresponds to a substring ω and is represented as a tuple $(id, len, posList, adjList)$. id is a unique identifier, len is the length of ω, $posList$ is a list of positions in S at which ω occurs and adjList is the list of successors of the node.

A node that corresponds to a substring that occurs only once in S is called a *uniqueNode*. If it appears multiple times it is called a *repeatNode*.

Ilia Minkin et al. (2016) introduce resembling definitions. A node in a de Bruijn graph that either has more than one incoming edge or has more than one outgoing edge or corresponds to either the first or last k-mer of an input string is called a junction. A path $u \rightsquigarrow v$ in a de Bruijn graph is called a non-branching path if besides u and v there are no junctions on this path. A non-branching path that cannot be extended is a maximal non-branching path. The graph obtained after compacting all maximal non-branching path into single edges is called a compacted de Bruijn graph. As seen in figure 2 the compacted and the compressed de Bruijn graph are not the same.

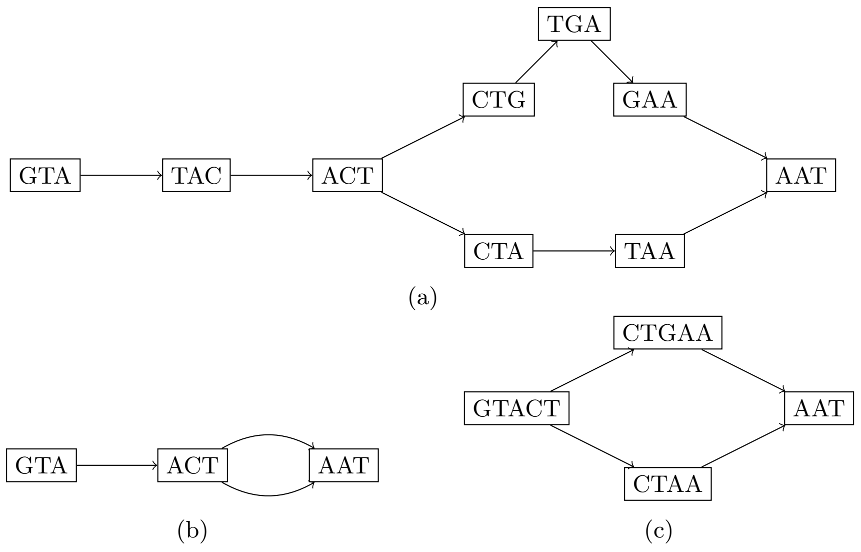


Fig. 2: $S = \{GTACTGAAT, GTACTAAT\}$ and $k = 3$. (a) The de Bruijn graph. (b) The compacted de Bruijn graph. (c) The compressed de Bruijn graph.

2.2 Other Data Structures

A suffix tree (ST) for a string S of length n is a rooted directed tree with n leaves numbered 1 to n (Gusfield 1997). Each edge is labeled with a nonempty substring of S. Concatenating the edge labels on the path from the root to leaf i spells out the i-th suffix of S ($S[i..n]$).

Given string S of length n, a suffix array (SA) for S is an array of integers that specifies the lexicographic ordering of the n suffixes of S (Gusfield 1997), i.e. $S_{SA[1]} \leq S_{SA[2]} \leq \ldots S_{SA[n]}$.

The Burrows-Wheeler transformation for a string S of length n is a string BWT of length n. It can be constructed with a $n \times n$ matrix of all rotations of S sorted lexicographically. The last column of that matrix forms the string BWT (Burrows and Wheeler 1994). The BWT can be constructed from the SA with $BWT[i] = S[SA[i] - 1]$ for all i with $SA[i] \neq 1$ and $BWT[i] = \$$ otherwise.

In the matrix of lexicographically sorted rotations of a string S we call the first column F and the last column L. L is equivalent to the BWT, F to the original string S. The LF-mapping is an array of integers such that $F[LF[i]] = L[i]$. It can be used to reconstruct S from the BWT. LF can be computed as $LF[i] = C[L[i]] + r_i$, where $C[c]$ counts the number of occurences of lexicographically smaller than c characters in S and r_i is the number of occurences of character $L[i]$ in the prefix $L[1..i]$ (Ferragina and Manzini 2000). The LF-mapping can also be computed from the SA. With $SA[i] = q$ and $SA[j] = q - 1$, $LF[i] = j$ ($LF[i] = 1$ if $SA[i] = 1$).

A substring R in a string S is called a repeat if it occurs at least twice in S. A repeat is called left-maximal if for all pairs $[i_1, j_1]$ and $[i_2, j_2]$ of occurences of R $S[i_1 - 1] \neq S[i_2 - 1]$. It is called right-maximal if for all pairs $[i_1, j_1]$ and $[i_2, j_2]$ of occurences of R $S[j_1 + 1] \neq S[j_2 + 1]$. A repeat that is left-maximal and right-maximal is called a maximal repeat (Abouelhoda et al. 2002).

A Bloom Filter is a probabilistic data structure (Bloom 1970). It supports insertion and testing if an element is a member of a set. It needs less memory than traditional set data structures but can produce false-positives. A Bloom Filter consists of a bit vector B, initialized to zeros and k independent hash functions $h_1, \ldots, h_k$. Inserting Element e is achieved by setting $B[h_i(e)] = 1$ for $1 \leq i \leq k$. An Element e is member of the set if for all $1 \leq i \leq k$ $B[h_i(e)] = 1$. The probability that a bit is 0 after n insertions is $\left(1 - \frac{1}{m}\right)^{kn}$. The probability for a false positive P_{fp} is the probability that all k bits are set to 1 after n insertions.

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{1}$$

3 Methods

The algorithms developed in Baier et al. (2016) are looking for positions at which nodes have to be split. All other nodes can be merged. Lemma 1 provides a criterion for splitting nodes.

Lemma 1 (Baier et al. 2016). *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.*

Per definition of the compressed de Bruijn graph u and v can only be merged if v is the only successor of u and u the only predecessor of v. So with lemma 1 if the string corresponding to v is a left-maximal repeat, v has two different predecessors and u and v need to be split. Similarly if the string corresponding to u is a right-maximal repeat, u has two different successors and u and v need to be split. Figure 3 illustrates this concept.

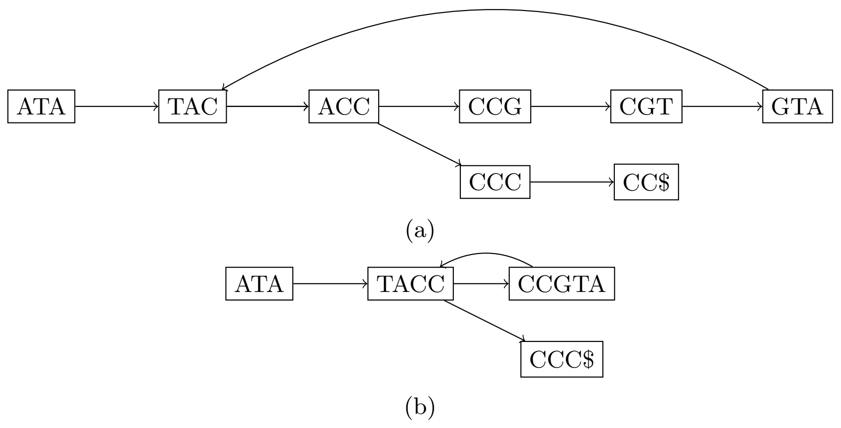


Fig. 3: (a) Illustration of Lemma 1 for string S=ATACCGTACCC$. TAC is a left-maximal repeat and has two different predecessors. ACC is a right-maximal repeat and has two different successors. (b) The resulting compressed de Bruijn graph. Note that since TAC has two different predecessors it is not merged with any of its predecessors. Similarly ACC is not merged with any successor since it has two different successors.

3.1 splitMEM and CST based Algorithm

I will briefly outline the splitMEM algorithm and the compressed suffix tree (CST) based algorithm (cst-based). For details refer to Marcus et al. (2014) and Baier et al. (2016). Both algorithms consist of the following two phases:

1. Compute set of repeatNodes
2. Compute uniqueNodes and edges

splitMEM computes the repeatNodes using a ST and "suffix skips" in $O(n\ log\ g)$ (n: total sequence length, g: length of longest genome). cst-based also uses a ST but runs in $O(n)$ time. The second step is achieved by sorting the starting positions of the repeatNodes, traversing the sorted list and adding uniqueNodes and edges along gaps between repeatNodes. cst-based uses a non-comparison-based sorting algorithm to achieve overall linear running time.

3.2 BWT Based Algorithm

Algorithm 1 uses the Burrows Wheeler Transformation and the LF-mapping to compute the compressed de Bruijn graph in a single backwards pass over S. In each iteration it either increments the current nodes length or splits the current node *cur* if a left-maximal or right-maximal repeat was found. Splitting *cur* at index p is done by adding p to $G[cur].posList$, computing the unique identifier *number* for the next node, adding *cur* to $G[number].adjList$, initializing $G[number].len$ to k and setting *number* as the current node. Lemma 1 provides a criterion for deciding if a split occurs. With iteration index p the currently processed suffix is $\omega = S_p$. Let $c = S[p-1]$. 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 ω must be split.

The naive approach for determining wether or not a substring is a left-maximal repeat, a right-maximal repeat or none of those would be very inefficient. Preprocessed bit vectors will be used to make these decisions in constant time. Querying these bit vectors requires the usage of the suffixes index in the SA. The first processed suffix is $S_p = \$$. Its SA index is 1 as \$ is lexicographically smaller than all other characters. So Algorithm 1 initializes $j \leftarrow 1$. The SA index for the next suffix S_{p-1} can then be obtained from the LF-mapping as $i \leftarrow LF(j)$.

For each right-maximal k-mer α a SA intervall $[lb, rb]$ can be obtained such that all suffixes starting with α and no other lie within this intervall in the SA. Bit vector B_1 is then initialized with zeros and for each right-maximal k-mer's SA intervall $[lb, rb]$ $B_1[lb]$ and $B_1[rb]$ are set to 1. Test (i) can than be performed in constant time by checking if either $B_1[i] = 1$ or $rank_1(B_1, i)$ is odd. $rank_1(B, i)$ return the number ones in B up to and including i and can be queried in constant time if precomputed. The next nodes unique identifier can be computed as $\lfloor (rank_1(B_1, i) + 1)/2 \rfloor$. Another bit vector B_2 is initialized to zeros. For each left-maximal k-mer $B_2[q]$ is set to 1 for all q with $lb \leq q \leq rb$. Test (ii) is than equivalent to asking $B_2[j] = 1$. A third bit vector B_3 is than used to determine a unique identifier for the new node. For each q with $B_2[q] = 1$, $B_3[LF(q)]$ is set to 1. Then for those indices i that correspond to a right-maximal k-mer (i.e. $B_1[i] = 1$ or $rank_1(B_1, i)$ is odd) $B_3[i]$ is reset to zero. The unique identifier is than set to $rightMax + rank_1(B_3, i-1) + 1$ with $rightMax = rank_1(B_1, n)/2$.

```
Algorithm 1: BWT-BASED
    Input: k, BWT, LF
    Output: Compressed de Bruijn Graph
 1  $(B_1, B_2, B_3) \leftarrow CREATE-BIT-VECTORS(k, BWT)$
 2  $rightMax \leftarrow rank_1(B_1, n)/2$
 3  $leftMax \leftarrow rank_1(B_3, n)$
 4  create array G of size $rightMax + leftMax + 1$
 5  $j \leftarrow 1$
 6  $cur \leftarrow rightMax + leftMax + 1$
 7  $G[cur].len \leftarrow 1$
 8  for $p \leftarrow n$ downto 2 do
 9      $i \leftarrow LF(j)$
10      $ones \leftarrow rank_1(B_1, i)$
11      $number \leftarrow \perp$
12      $c \leftarrow BWT[i]$
13      if (i): $B_1[i] = 1$ or ones is odd then
14          $number \leftarrow \lfloor (ones + 1)/2 \rfloor$
15      else if (ii): $B_2[j] = 1$ then
16          $number \leftarrow rightMax + rank_1(B_3, i - 1) + 1$
17      if $c = \#$ then
18          add p to front of $G[cur].posList$
19          add new node to G
20          $cur \leftarrow G.size$
21          $G[cur].len \leftarrow 1$
22      else if $number \neq \perp$ then
23          add p to front of $G[cur].posList$
24          add cur to front of $G[number].adjList$
25          $G[number].len \leftarrow k$
26          $cur \leftarrow number$
27  add 1 to $G[cur].posList$
```

The bit vectors can be constructed using the longest common prefix array (LCP) in $O(n \log \sigma)$. The LF-mapping is implemented using a wavelet tree and thus requires $O(\log \sigma)$ time. All other operations take constant time. So the overall running time is in $O(n \log \sigma)$.

3.3 TwoPaCo

Ilia Minkin et al. (2016) present a scalable, low-memory algorithm (TwoPaCo) that constrcuts the compacted de Bruijn graph. Compaction of the de Bruijn graph is reduced to the problem of finding junctions. The basic Filter Junction algorithm (Algorithm 2) called with an ordinary set data structure finds these junctions but uses large amounts of memory. Instead it is called with a bloom filter as a set data structure. This drastically reduces memory consumption but can return false-positives. To eliminate these false-positives a second round is called with a hash table. As the first round significantly reduced the candidate set the memory footprint of the hash table is largely reduced. To further reduce the memory footprint the set of input strings can be partitioned and processed in seperate rounds.

Lemma 2 states that there is a bijunction between maximal non-branching paths and junctions of the de Bruijn graph. So in TwoPaCo the set of junctions is computed, which implies the compacted de Bruijn graph.

Lemma 2 (Ilia Minkin et al. 2016). *Let s be an input string and P be the set of maximal non-branching paths of the graph $G(S,k)$. Let T be the set of substrings of s such that each $t \in T$ starts and ends with a junction of $G(s,k)$ and does not contain junctions in between. Then there exists a bijective function $g : T \rightarrow P$.*

Algorithm 2 is a naive algorithm for finding junctions. First it calculates the set of edges E in the de Bruijn graph based on the set of candidate junctions (lines 1 to 5). E is then used to count the number of incoming and outgoing edges. If there is only one incoming and one outgoing edge then the corresponding node is not a junction and is thus unmarked.

Algorithm 2: Filter Junctions

```
Input: strings $S = \{s_1, \dots, s_n\}$, integer k, empty set E, candidate set of
       marked junction positions $C \supseteq J(S,k)$
Output: reduced set C
1  for $s \in S$ do
2      for $1 \leq i < |s| - k$ do
3          if $C[s,i] = marked$ then
4              insert $s[i..i+k]$ into E
5              insert $s[i-1..i-1+k]$ into E
6  for $s \in S$ do
7      for $1 \leq i < |s| - k$ do
8          $v \leftarrow s[i..i+k-1]$
9          if $C[s,i] = marked$ and $v$ is not first or last k-mer of $s$ then
10             $in \leftarrow 0$
11             $out \leftarrow 0$
12             for $c \in \{A, C, G, T\}$ do
13                 if $v \cdot c \in E$ then
14                     $out \leftarrow out + 1$
15                 if $c \cdot v \in E$ then
16                     $in \leftarrow in + 1$
17             if $in = 1$ and $out = 1$ then
18                 $C[s,i] \leftarrow unmarked$
19 return $C$
```

Algorithm 3 calls algorithm 2 twice. In the first pass a Bloom Filter is used to store E. The Bloom Filter can lead to false-positives. Because of that the edge counters *in* and *out* might be greater than their actual value in the de Bruijn graph thus leaving positions that should be unmarked in C as marked. In the second pass E is stored in a hash table. As the candidate set of junction positions is significantly reduced after the first pass the amount of edges that need to be stored in E is significantly smaller as well. With E stored in a hash table no false-positives occur and the returned set C corresponds exactly to the set of junction positions in the de Bruijn graph.

```
Algorithm 3: Filter Junctions Two-Pass
  Input: strings $S = \{s_1, \ldots, s_n\}$, integer k, candidate set of junction
         positions $C_{in}$, integer b
  Output: candidate set of junction positions $C_{out}$
1 $F \leftarrow$ empty Bloom Filter of size b
2 $C_{temp} \leftarrow Filter - Junctions(S, k, F, C_{in})$
3 $H \leftarrow$ empty hash table
4 $C_{out} \leftarrow Filter - Junctions(S, k, H, C_{temp})$
5 return $C_{out}$
```

For large inputs the hash table in algorithm 3 might still not fit into the main memory. So in algorithm 4 (TwoPaCo) the input is partitioned into l parts and algorithm 3 is run l times with each round processing one part. The partitioning is achieved by dividing all input k-mers. It is important that for a k-mer that occurs multiple times at different positions all positions are handled in the same round. In round i only the positions of the k-mers in partition i are initialized as marked in the cadidate set of junction positions C_i. This leads to a smaller set of edges E in algorithm 2 and thus decreases memory consumption.

```
Algorithm 4: TwoPaCo
  Input: strings $S = \{s_1, \ldots, s_n\}$, integer k, integer l, integer b
  Output: compacted de Bruijn graph
1 $C_{init} \leftarrow$ boolean array (initialized as unmarked)
2 Divide k-mers of S into l partitions
3 for $0 \leq i < l$ do
4     $C_i \leftarrow$ mark every position of $C_{init}$ belonging to partion i
5     $C'_i \leftarrow FilterJunctionsTwoPass(S, k, b, C_i)$
6 $C_{final} \leftarrow \bigcup C'_i$
7 return Graph implied by $C_{final}$
```

4 Results

Marcus et al. (2014) implemented splitMEM in C++. Baier et al. (2016) implemented their algorithms cst-based and bwt-based in C++ using sdsl (Gog et al. 2014). Ilia Minkin et al. (2016) implemented TwoPaCo in C++ using Intel Thread Building Blocks (TBB).

Baier et al. (2016) performed experiments on 40 E.coli genomes, on 62 E.coli genomes (list of genomes in Supplementary Material of Marcus et al. (2014)) and on 7 human genomes. "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." Table 1 shows running times and peak memory consumption. Clearly cst-based and bwt-based outperform splitMEM significantly. They are also able to compute the compressed de Bruijn graph for seven human genomes which was not possible with splitMEM as the memory requirements were too high. Table 2 shows some statistics about the calculated compressed de Bruijn graphs. The graph size is shown in bytes per base pair. Storing the sequences in an unstructerd way requires 0.25 byte per base pair (as characters $\{A, C, G, T\}$ can be encoded with 2 bits). As seen in the table the storage size for the compressed de Bruijn graph can go as high as 1.65 bytes per base pair (6.6 times more than storing sequences) and as low as 0.06 (4.33 times less). The higher storage needs come from overlapping characters in nodes. Storage is saved with repeatNodes as they only need to be saved once. With greater k the graph gets smaller. The table also shows that there are more repeatNodes than uniqueNodes, especially for smaller values of k. An interesting difference between E.coli and humans is that repeatNodes are larger than uniqueNodes in humans (for $k = 50$ and $k = 100$) and smaller in E.coli.

Table 1: Benchmark taken from Baier et al. (2016). Cells show running times in seconds and in parentheses the maximum main memory usage in bytes per base pair.

	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)
7 HG			
splitMEM	-	-	-
cst-based	87605 (4.74)	82812 (4.62)	80116 (4.58)
bwt-based	29014 (2.78)	28129 (2.22)	28588 (2.05)

Ilia Minkin et al. (2016) ran their experiments "on the highest memory Amazon EC2 instance (r3.8xlarge): a server with Intel Xeon E5-2670 processors and 244 GB of RAM". They used the same set of 62 E.coli genomes and 7 human genomes to compare splitMEM, bwt-based and TwoPaCo. They ran bwt-based with a single strand and both strands (added reverse complement of inputs to sequence). They ran TwoPaCo with a single thread and 15 threads. Table 3 shows that bwt-based outperforms splitMEM (around 8-9 times faster, around 209-356 times less memory). It also shows that TwoPaCo outperforms bwt-based

Table 2: Statistics about the compressed de Bruijn graphs taken from Baier et al. (2016). Graph size is shown in bytes per base pair.

	k=50	k=100	k=1000
62 E.coli (310 million base pairs)			
graph size	1.12	0.68	0.06
edges	16304084	9219061	555810
nodes	1007765	738980	117021
uniqueNodes	174717	141167	34463
repeatNodes	833048	597813	82558
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
7 HG (21201 million base pairs)			
graph size	1.65	1.16	1.00
edges	2056675301	1475958859	1319219774
nodes	25367105	12030826	3851688
uniqueNodes	2614834	2316797	1143848
repeatNodes	22752271	9714029	2707840
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%

(around 1.9-45 times faster and 2.6-44 times less memory based on configuration and parameters).

Table 3: Benchmark taken from Ilia Minkin et al. (2016). Cells show running time in minutes and memory usage (gigabytes) in parentheses. A dash indicates that the program ran out of memory. splitMEM and bwt-based were run using standard paramters. TwoPaCo was run using one round and Bloom filter sizes $b = 0.13GB$ for E.coli, $b = 4.3GB$ for human with k = 25 and $b = 8.6GB$ for human with k = 100.

	splitMEM (Marcus et al. 2014)	bwt-based (Baier et al. 2016)		TwoPaCo (Ilia Minkin et al. 2016)	
	single strand	single strand	both strands	1 thread	15 threads
E.coli (k=25)	70 (178.0)	8 (0.85)	12 (1.7)	4 (0.16)	2 (0.39)
E.coli (k=100)	67 (178.0)	8 (0.5)	12 (1.0)	4 (0.19)	2 (0.39)
7 humans (k=25)	-	867 (100.3)	1605 (209.88)	436 (4.4)	36 (4.84)
7 humans (k=100)	-	807 (46.02)	1080 (92.26)	317 (8.42)	57 (8.75)

5 Discussion

New sequencing techniques produce large amounts of genomic sequences, including sequences of individuals of the same species. Analyzing the structural similarities and differences of those sequences is a difficult task. The compressed de Bruijn graph is a suitable data structure for the pan-genome. Baier et al. (2016) introduced bwt-based, an algorithm that is significantly better than its competitors but still was not able to compute the compressed de Bruijn graph for more than 7 human genomes on a machine with 128GB RAM. The peak memory consumption occurs during the construction of the BWT. Ilia Minkin et al. (2016) introduced TwoPaCo. TwoPaCo does not use any suffix data structure, instead uses a Bloom Filter and partitioning to adapt to the available memory. Not only is this algorithm faster, it allows to run significantly bigger sets of input strings. TwoPaCo was used with inputs of up to 100 human genomes.

As seen in figure 2 the compacted de Bruijn graph calculated by TwoPaCo is not the same as the compressed de Bruijn graph. So the benchmarks are questionable. TwoPaCo needs to be modified in a way that it calculates the compressed de Bruijn graph. Its core is the Filter-Junction algorithm. Junctions are almost the same as repeatNodes. If it would not take into account wether or not the substring is the first or last k-mer of an input string, the positions of the repeatNodes would be calculated. These could then be used with step 2 of the cst-based algorithm to calculate the compressed de Bruijn graph.

References

Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch. "The enhanced suffix array and its applications to genome analysis". In: *International Workshop on Algorithms in Bioinformatics*. Springer. 2002, pp. 449–463.

Baier, Uwe, Timo Beller, and Enno Ohlebusch. "Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform". In: *Bioinformatics* 32.4 (2016), pp. 497–504. DOI: `10.1093/bioinformatics/btv603`. eprint: `http://bioinformatics.oxfordjournals.org/content/32/4/497.full.pdf+html`. URL: `http://bioinformatics.oxfordjournals.org/content/32/4/497.abstract`.

Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

Burrows, Michael and David J Wheeler. "A block-sorting lossless data compression algorithm". In: (1994).

Consortium, 1000 Genomes Project et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (2015), pp. 68–74.

Ferragina, Paolo and Giovanni Manzini. "Opportunistic data structures with applications". In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on.* IEEE. 2000, pp. 390–398.

Gog, Simon et al. "From theory to practice: Plug and play with succinct data structures". In: *International Symposium on Experimental Algorithms.* Springer. 2014, pp. 326–337.

Gusfield, Dan. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press, 1997.

Marcus, Shoshana, Hayan Lee, and Michael C Schatz. "SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips". In: *Bioinformatics* 30.24 (2014), pp. 3476–3483.

Minkin, Ilia, Son Pham, and Paul Medvedev. "TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes". In: *arXiv preprint arXiv:1602.05856* (2016).

Minkin, Ilya et al. "Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes". In: *International Workshop on Algorithms in Bioinformatics.* Springer. 2013, pp. 215–229.

Raphael, Benjamin et al. "A novel method for multiple alignment of sequences with repeated and shuffled elements". In: *Genome Research* 14.11 (2004), pp. 2336–2346.