# Generating Multiple Alignments of Genomes of the Same Species

Jannik Olbrich[a], Thomas Büchler[b] and Enno Ohlebusch[c]
*Institute of Theoretical Computer Science, Ulm University, Germany*

Keywords: Multiple Genome Alignment, Pangenome, Prefix-Free Parsing, Maximal Unique Matches, Suffix Array.

Abstract: In this paper, we tackle the problem of generating a multiple alignment of assembled genomes of individuals of the same species. Of course, a (colinear) multiple alignment cannot capture structural variants such as inversions or transpositions, but if these are relatively rare (as, for instance, in human or mouse genomes), it makes sense to generate such a multiple alignment. In the following, it is assumed that each assembled genome is composed of contigs. We will show that the combination of a well-known anchor-based method with the technique of prefix-free parsing yields an approach that is able to generate multiple alignments on a pangenomic scale, provided that large structural variants are rare. Furthermore, experiments with real world data show that our software tool PANAMA (PANgenomic Anchor-based Multiple Alignment) significantly outperforms current state-of-the art programs.

## 1 INTRODUCTION

A single linear reference is commonly used to represent the human genome in genomic studies and diagnostics. However, there are a lot of differences between the genomes of individuals of the same species and a single reference is unable to cover them. (The 1000 Genomes Project Consortium, 2015) produced a database that includes the genomes of 2504 different humans and the differences between them. Including common variations in the reference gives a more accurate representation of the genomes of a species. We call such a representation the pangenome of the population. It should be pointed out that throughout this paper we use the term "pangenome" in a narrower sense. In a broader sense, the pangenome defines the entire genomic repertoire of a given phylogenetic clade, which may range from species to phylum and beyond. Note that (Tettelin et al., 2005) coined the term pangenome two decades ago; they evaluated the composition of six strains of *Streptococcus agalactiae*.

A pangenome is often constructed from a reference sequence and a VCF-file containing the variations. Since novel long read sequencing technologies allow for de novo assembly of many individuals of a species or population (Porubsky et al., 2021), high-quality assemblies are becoming widely available. In this paper, we tackle the problem of generating a multiple alignment of assembled genomes of many individuals of the same species. Of course, a (colinear) multiple alignment cannot capture structural variants such as inversions or transpositions, but in many species these are relatively rare. (In future work, we intend to combine our method with techniques that detect such structural variants.) Up to now, the computation of such a chromosome-scale multiple alignment was not possible because no multiple aligner was able to deal with such a volume of data (cf. Section EXPERIMENTAL RESULTS).

(Höhl et al., 2002) presented MGA (Multiple Genome Aligner), the first software-tool that was able to compute a multiple alignment of closely related genomes. However, MGA was limited to viruses and strains of bacteria (Chain et al., 2003). MGA and many other software-tools for aligning multiple genomic DNA sequences use an anchor-based method that is composed of three phases:

1. computation of fragments (segments in the genomes that are identical or very similar),
2. computation of a highest-scoring global chain of colinear non-overlapping fragments: these are the anchors that form the backbone of the alignment,
3. alignment of the regions between the anchors (either by applying the same method recursively or by applying a different multiple sequence alignment program).

[a] https://orcid.org/0000-0003-3291-7342
[b] https://orcid.org/0000-0002-9273-5439
[c] https://orcid.org/0009-0008-3937-3652



Olbrich, J., Büchler, T. and Ohlebusch, E.
Generating Multiple Alignments of Genomes of the Same Species.
DOI: 10.5220/0013165400003911

In *Proceedings of the 18th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2025) - Volume 1*, pages 459-468
ISBN: 978-989-758-731-3; ISSN: 2184-4305

A recent tool that elaborates on this anchor-based method is FMAlign2 (Zhang et al., 2024). However, FMAlign2 is still limited to datasets of a few hundred million bases (cf. Section EXPERIMENTAL RESULTS).

In this paper, we will show that it is possible to generate a multiple alignment of a set of assembled genomes (of the same species), where an assembled genome is set of contigs. Our new method is depicted in Figure 1. We combine MGA's anchor-based method with the technique of prefix-free parsing (PFP), which was introduced by (Boucher et al., 2019). This technique parses a DNA sequence (a chromosome composed of contigs or a complete chromosome) $S$ into phrases, and two phrases have the same identifier (meta-symbol) if and only if they are identical on the base-level. Thus, the parse $P$ is the sequence of identifiers that gives $S$ if each identifier is replaced with its phrase. The main idea is to first compute anchors on the parse $P$ instead of computing anchors on $S$, see Figure 1. Given $m$ DNA sequences from the same chromosome of different individuals (e.g. chromosome 19 of 1000 different humans) as input, our method uses the following phases, which are explained in subsequent sections:

1. Compute the parses of the $m$ sequences
2. On the parses, compute the backbone of the overall multiple alignment as follows:
   (a) Compute the generalized suffix array
   (b) Find `multiMUMs` (the fragments)
   (c) Compute anchors by chaining `multiMUMs`
   (d) Extend anchors on the base-level
3. Generate an alignment of the gaps[1]
   (a) While there remains a large gap, apply Phases 2a-2c to the gap and add the fragments in the resulting chain as anchors (here, `multiMUMs` may be partial)
   (b) Generate an alignment of the remaining (small) gaps using FMAlign2 and MAFFT (Nakamura et al., 2018)

In this way, we are able to generate multiple alignments on a pangenomic scale. It should be pointed out that our new method can only be successful if the chromosomes of individuals from the same species are very similar DNA sequences; in particular structural chromosomal rearrangements must be rare. Therefore, our software tool PANAMA is a special purpose multiple sequence alignment program. On the other hand, there is an urgent need for such a program (Liao et al., 2023). Experiments with real world data show that our program PANAMA outperforms current state-of-the art programs.

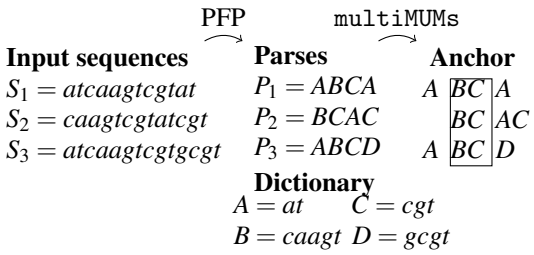


Figure 1: First, the parses of the input sequences will be calculated. Then anchors are determined within the parses.

## 2 PREFIX-FREE PARSING

Prefix-free parsing is a technique invented by (Boucher et al., 2019). In the simple version used here, it uses a rolling hash (a hash function where the input is hashed in a window that moves through the input) to divide a string $S$ into substrings, which form the dictionary $D$. The name prefix-free parsing is justified by the property that no suffix of a string from $D$ is a proper prefix of a suffix of any other string from $D$ (but this property does not play a role in our context). In the following more detailed explanation, we assume the reader to be familiar with the Karp-Rabin-Algorithm, see e.g. (Cormen et al., 1990). This algorithm uses a sliding window of fixed size $w$ and a hash function KR. For every position $i$ in $S$, it computes the hash value $\mathrm{KR}(W_i)$ of the substring $W_i = S[i..i+w-1]$. Since the hash value of $W_{i+1}$ can be computed in constant time from the previous hash value, the parsing algorithm takes only linear time. Given a fixed number $p$ (called modulus[2]), the string $W_i$ is called a trigger string if and only if $\mathrm{KR}(W_i) \bmod p = 0$. In a left-to-right scan of $S$, the parsing algorithm breaks $S$ into substrings so that each substring ends with a trigger string (and contains no other trigger string). This gives the dictionary $D$ and the parse $P$. In $P$, phrases (elements of $D$) are represented by their lexicographic rank. More precisely, the phrases are ordered lexicographically and every phrase is identified with its rank in the sorted dictionary (i.e., the ranks serve as meta-symbols). Consequently, the parse $P$ is the sequence of numbers that gives the string $S$ if each number is replaced with its phrase from $D$. In the example of Figure 1, the win-

[1]For brevity, we will use the term 'gap' instead of 'region between anchors' from now on (it should not be confused with a gap in an alignment).

[2]In the Karp-Rabin-Algorithm, the modulus is a prime number, but this is not required in our context.

dow size is $w = 1$, '$t$' is used as trigger string, and upper case letters are used as meta-symbols.

# 3 PRELIMINARIES (FOR COMPUTING multiMUMs)

Let $S$ be a string of length $n$ on an ordered alphabet $\Sigma$. For $1 \le i \le n$, $S[i]$ denotes the *character at position i* in $S$. For $i \le j$, $S[i..j]$ 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* $\mathsf{SA}$ 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 $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$. The suffix array can be built in linear time; we refer to the overview article of (Puglisi et al., 2007) for suffix array construction algorithms and to (Olbrich et al., 2024) for newer developments.

Let $S$ be a string of length $n$ having the sentinel character \$ at the end (and nowhere else). We assume that \$ is smaller than any other character. The Burrows and Wheeler transform (Burrows and Wheeler, 1994) converts $S$ into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i]-1]$ for all $i$ with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise.

The suffix array $\mathsf{SA}$ is often enhanced with the so-called $\mathsf{LCP}$-array containing the lengths of longest common prefixes between consecutive suffixes in $\mathsf{SA}$. Formally, the $\mathsf{LCP}$-array is an array so that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n+1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \le i \le n$, where $\mathsf{lcp}(u,v)$ denotes the longest common prefix between two strings $u$ and $v$. Like the suffix array, the $\mathsf{LCP}$-array can be computed in linear time (Kasai et al., 2001). (Abouelhoda et al., 2004) introduced the concept of lcp-intervals. An interval $[i..j]$, where $1 \le i < j \le n$, in the $\mathsf{LCP}$-array is called an *lcp-interval of lcp-value* $\ell$ (denoted by $\ell$-$[i..j]$) if

- $\mathsf{LCP}[i] < \ell$,
- $\mathsf{LCP}[k] \ge \ell$ for all $k$ with $i+1 \le k \le j$,
- $\mathsf{LCP}[k] = \ell$ for at least one $k$ with $i+1 \le k \le j$,
- $\mathsf{LCP}[j+1] < \ell$.

(Abouelhoda et al., 2004) showed that there is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the suffix tree of $S$ (we assume a basic knowledge of suffix trees). Consequently, there are at most $n-1$ lcp-intervals for a string of length $n$.

Let $S^1, S^2, \ldots, S^m$ be strings of sizes $n_1, n_2, \ldots, n_m$, respectively. We are interested in the lexicographic order of all suffixes

$$S^1_1, \ldots, S^1_{n_1}, S^2_1, \ldots, S^2_{n_2}, \ldots, S^m_1, \ldots, S^m_{n_m}$$

of these strings. Note that two suffixes $S^j_p$ and $S^k_q$ with $j \neq k$ may coincide, i.e., $S^j_p = S^k_q$ is possible. (In this case, it is natural to demand that the suffix with the smaller superscript shall appear before the suffix with the larger superscript.) Because the strings may share identical suffixes, we use $m$ pairwise distinct characters $\#_1, \#_2, \ldots, \#_m$ to tell the suffixes apart. To be precise, for each $j$ with $1 \le j \le m$, we obtain the string $S^j\#_j$ of length $n_j+1$ by appending the special character $\#_j$ to $S^j$. This ensures that each suffix can uniquely be assigned to one of the $m$ strings: if the suffix ends with $\#_j$, then it belongs to $S^j$. If we assume that $\#_1 < \#_2 < \cdots < \#_m$ and that all other characters in the alphabet $\Sigma$ are larger than these symbols, then the suffixes of the strings $S^1\#_1, S^2\#_2, \ldots, S^m\#_m$ are not only pairwise distinct, but we also have $S^j_p\#_j < S^k_q\#_k$ if and only if either $S^j_p < S^k_q$ or $S^j_p = S^k_q$ and $j < k$. In the following, we tacitly assume that every string $S^j$ ($1 \le j \le m$) is terminated with the character $\#_j$, so its size is $n_j+1$.

The *generalized suffix array* (GSA) of the strings $S^1, S^2, \ldots, S^m$ consists of *two* arrays of size $n = m + \sum_{j=1}^{m} n_j$, the *document array* $\mathsf{DA}$ and the array $\mathsf{SA}$, having the following properties:

- For every suffix $S^j_k$, there is an index $i$ so that $j = \mathsf{DA}[i]$ and $k = \mathsf{SA}[i]$.
- $S^{\mathsf{DA}[i]}_{\mathsf{SA}[i]} < S^{\mathsf{DA}[i+1]}_{\mathsf{SA}[i+1]}$ for all $i$ with $1 \le i \le n-1$.

In other words, the arrays $\mathsf{DA}$ and $\mathsf{SA}$ specify the lexicographic order of all the suffixes of the $m$ strings. An example can be found in (the two rightmost columns of) Figure 2, which also shows the corresponding $\mathsf{LCP}$-array and $\mathsf{BWT}$. We will call the combination of the GSA with the $\mathsf{LCP}$-array the *enhanced* GSA of $S^1, S^2, \ldots, S^m$. (Louza et al., 2017) have shown that the (enhanced) GSA can be constructed in $O(n)$ time with only one special character instead of the $m$ special characters $\#_1, \#_2, \ldots, \#_m$ (this is advantageous because it keeps the alphabet small). However, it is conceptually easier to use $m$ special characters instead of one.

# 4 COMPUTATION OF multiMUMs

In order to determine anchors (on the parse-level or on the base-level), we must first compute fragments and then a highest-scoring global chain of colinear non-overlapping fragments. MGA (Höhl et al., 2002) as well as FMAlign2 (Zhang et al., 2024) use `multiMEMs` as fragments, but like parsnp (Treangen

| $i$ | LCP | BWT | $S^{\mathsf{DA}[i]}_{\mathsf{SA}[i]}$ | DA | SA |
|---|---|---|---|---|---|
| 1 | $-1$ | $A$ | $\#_1$ | 1 | 5 |
| 2 | 0 | $C$ | $\#_2$ | 2 | 5 |
| 3 | 0 | $D$ | $\#_3$ | 3 | 5 |
| 4 | 0 | $C$ | $A\#_1$ | 1 | 4 |
| 5 | 1 | $\#_1$ | $ABCA\#_1$ | 1 | 1 |
| 6 | 2 | $\#_3$ | $ABCD\#_3$ | 3 | 1 |
| 7 | 1 | $C$ | $AC\#_2$ | 2 | 3 |
| 8 | 0 | $A$ | $BCA\#_1$ | 1 | 2 |
| 9 | 3 | $\#_2$ | $BCAC\#_2$ | 2 | 1 |
| 10 | 2 | $A$ | $BCD\#_3$ | 3 | 2 |
| 11 | 0 | $A$ | $C\#_2$ | 2 | 4 |
| 12 | 1 | $B$ | $CA\#_1$ | 1 | 3 |
| 13 | 2 | $B$ | $CAC\#_2$ | 2 | 2 |
| 14 | 1 | $B$ | $CD\#_3$ | 3 | 3 |
| 15 | 0 | $C$ | $D\#_3$ | 3 | 4 |
| 16 | $-1$ | | | | |

Figure 2: The GSA of the strings $S^1 = ABCA\#_1$, $S^2 = BCAC\#_2$, and $S^3 = ABCD\#_3$ (derived from the parses of Figure 1) consists of the two arrays DA and SA. The enhanced GSA includes the corresponding LCP-array (note that the BWT can easily be computed on the fly).

et al., 2014) we use `multiMUMs` instead. This is because the explanation of how `multiMUMs` can be computed is much easier to understand and preliminary experiments showed that `multiMUMs` are equally good. Roughly speaking, a `multiMUM` is a string $\omega$ occurring exactly once in each of the sequences $S^1,\ldots,S^m$ with the property that $\omega$ cannot simultaneously be extended in all sequences (on either end) without incurring a mismatch. The formal definition reads as follows.

**Definition 1.** *A multiple exact match in the sequences $S^1,\ldots,S^m$ is an $(m+1)$-tuple $(\ell,p_1,\ldots,p_m)$ with $\ell > 0$ and $1 \le p_k \le n_k - \ell + 1$ $(1 \le k \le m)$ so that $S^i[p_i..p_i+\ell-1] = S^j[p_j..p_j+\ell-1]$ for all $i,j \in \{1,\ldots,m\}$. In words, the length $\ell$ substrings of $S^1,\ldots,S^m$ starting at the positions $p_1,\ldots,p_m$ coincide. A multiple exact match is* left-maximal *if for at least one pair $(i,j)$ we have $S^i[p_i-1] \neq S^j[p_j-1]$ (for $k \in \{1,\ldots,m\}$, we define $S^k[p_k-1] = \#_k$ if $p_k = 1$, see the definition of the* BWT*). It is* right-maximal *if for at least one pair $(i,j)$ we have $S^i[p_i+\ell] \neq S^j[p_j+\ell]$. A multiple exact match is* maximal *if it is left-maximal and right-maximal. A multiple maximal exact match is also called* `multiMEM`*. A* `multiMEM` *$(\ell,p_1,\ldots,p_m)$ is a* `multiMUM` *(multiple maximal unique match) if for all $i$ with $1 \le i \le m$ the string $S^i[p_i..p_i+\ell-1]$ occurs exactly once in the sequence $S^i$.*

If there is a long `multiMUM` $(\ell,p_1,\ldots,p_m)$ in the sequences $S^1,\ldots,S^m$, then it is very likely that the identical substrings $S^1[p_1..p_1+\ell-1],\ldots,S^m[p_m..p_m+\ell-1]$ appear one below the other in a multiple alignment of $S^1,\ldots,S^m$. In other words, the `multiMUM` serves as a potential anchor. The next lemma tells us how `multiMUMs` can efficiently be computed.

**Lemma 2.** *There is a one-to-one correspondence between the set of all* `multiMUMs` *and the set of all lcp-intervals $\ell$-$[lb..rb]$ in the enhanced GSA of $S^1,\ldots,S^m$ satisfying*

*(1)* $rb - lb + 1 = m$.

*(2)* $\mathsf{DA}[i] \neq \mathsf{DA}[j]$ *for all pairs $(i,j)$ with $lb \le i < j \le rb$.*

*(3)* $\mathsf{BWT}[i] \neq \mathsf{BWT}[j]$ *for at least one pair $(i,j)$ with $lb \le i < j \le rb$.*

*Proof.* Let $\ell$-$[lb..rb]$ be an lcp-interval satisfying the three conditions. By conditions (1) and (2), we have $\{\mathsf{DA}[k] \mid 1 \le k \le m\} = \{1,\ldots,m\}$. That is, the $m$ suffixes in $[lb..rb]$ belong to $m$ different strings. Let $i_1,\ldots,i_m$ be the permutation of the indices $lb, lb+1,\ldots,rb$ so that $\mathsf{DA}[i_k] = k$ for $1 \le k \le m$. Define $p_k = \mathsf{SA}[i_k]$ for $1 \le k \le m$. We claim that $(\ell,p_1,\ldots,p_m)$ is a `multiMUM`. By the definition of an lcp-interval, the length $\ell$ string $\omega = S^k[p_k..p_k+\ell-1]$ is a common prefix of the suffixes in $[lb..rb]$. It follows that $\omega$ occurs exactly once in each of the strings $S^1,\ldots,S^m$ and that $(\ell,p_1,\ldots,p_m)$ is a multiple exact match. By condition (3), it is left-maximal. Since there is at least one index $q$ in $[lb..rb]$ so that $\mathsf{LCP}[q] = \ell$ (definition of lcp-interval), it is also right-maximal. In summary, $(\ell,p_1,\ldots,p_m)$ is a `multiMUM`.
Conversely, let $(\ell,p_1,\ldots,p_m)$ be a `multiMUM`. That is, the string $\omega = S^k[p_k..p_k+\ell-1]$ $(1 \le k \le m)$ occurs exactly once in each sequence $S^1,\ldots,S^m$. In combination with the right-maximality this implies that there is an lcp-interval $\ell$-$[lb..rb]$ that contains exactly the suffixes $S^1_{p_1},\ldots,S^m_{p_m}$. This lcp-interval satisfies conditions (1) and (2). It also satisfies condition (3) because $(\ell,p_1,\ldots,p_m)$ is left-maximal. □

According to the preceding lemma we can compute `multiMUMs` as follows: Enumerate all lcp-intervals and for each lcp-interval $\ell$-$[lb..rb]$ of size $m$ check whether the $m$ indices in $[lb..rb]$ satisfy conditions (2) and (3). In the example of Figure 2, 2-[8..10] is the only lcp-interval of size 3. This interval fulfills conditions (2) and (3) and hence it corresponds to the `multiMUM` $(2,2,1,2)$, which is the anchor in Figure 1.

It is well known that all lcp-intervals can be enumerated in $O(n)$ (Abouelhoda et al., 2004; Kasai et al., 2001). The enumeration algorithm is shown in Appendix 8. An alternative method for computing `multiMUMs` can be found in (Ohlebusch and Kurtz, 2008). It works by (a) separately streaming each string $S^j$ $(2 \le j \le m)$ against the suffix tree of $S^1$ and

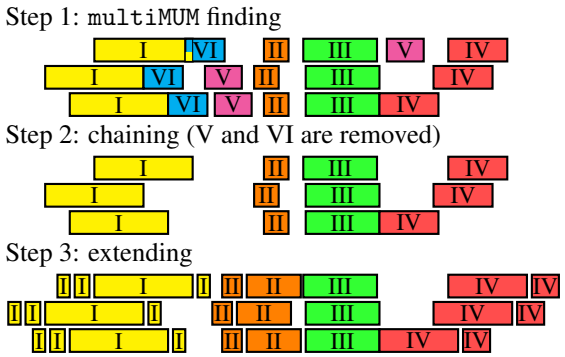


Figure 3: Schematic constructing of our backbone. Step 1 runs on the parse, while Step 3 is done on the base-level. Note that the situation of multiMUM V is highly unlikely to occur (especially on the parse-level) and would likely indicate a transposition. This case is included solely to exhaustively display the reasons why a multiMUM may not be included in the chain. Note that the Steps 1–3 here correspond to Phases 2b–2d (see Introduction).

(b) combining the pairwise exact matches to multiple exact matches. In fact, the method described in (Ohlebusch and Kurtz, 2008) computes rare multiMEMs, but it yields multiMUMs if the rareness-thresholds are all set to 1.

## 5 FRAGMENT-CHAINING AND EXTENSION OF ANCHORS

To find anchors, one must find a highest-scoring global chain of colinear non-overlapping fragments. In our context, a fragment $f$ is a multiMUM $(\ell, p_1, \ldots, p_m)$. We associate a weight with each fragment, denoted by $f.weight$. If $f$ is a multiMUM on the base-level, then we set $f.weight = \ell$. If $f$ is a multiMUM on the parse-level, then $f.weight$ is the number of bases that are obtained by replacing the identifiers (meta-symbols) in $S^k[p_k..p_k + \ell - 1]$ with their phrases from the dictionary.

Roughly speaking, two fragments $f$ and $f'$ are *colinear* if their order is the same in all of the sequences. In Step 1 of Figure 3, for example, the fragments I and II are colinear, but II and V are not. Two fragments *overlap* if their segments overlap in one of the sequences (in Figure 3, the fragments I and VI are overlapping, while I and II are non-overlapping).

We define a binary relation $\ll$ on the set of fragments so that $f \ll f'$ if and only if $f$ and $f'$ are colinear and non-overlapping.

**Definition 3.** *Let $f = (\ell, p_1, \ldots, p_m)$ and $f' = (\ell', p'_1, \ldots, p'_m)$ be two fragments. We define $f \ll f'$ if and only if $p_k + \ell - 1 < p'_k$ for all $k$ with $1 \le k \le m$. We then say that $f$* precedes *$f'$.*

*A chain $C$ of colinear non-overlapping fragments ('chain' for short) is a sequence of fragments $f_1, f_2, \ldots, f_j$ so that $f_i \ll f_{i+1}$ for all $i$ with $1 \le i < j$. The* score *of $C$ is $score(C) = \sum_{i=1}^{j} f_i.weight$.*
*Given $m$ weighted fragments, the* global fragment-chaining problem *is to determine a chain of highest score (called* optimal global chain *in the following) starting at the origin* 0 *and ending at terminus* t*. (The origin $0 = (0, 0, \ldots, 0)$ and the terminus $t = (0, n_1 + 1, \ldots, n_m + 1)$ are artificial fragments with weight 0. Note that $0 \ll f \ll t$ for every fragment $f$ with $f \neq 0$ and $f \neq t$.)*

Let $f'.score$ be defined as the maximum score of all chains starting at 0 and ending at $f'$. Then $f'.score$ can be expressed by the recurrence: $0.score = 0$ and

$$f'.score \quad = \quad f'.weight + \max\{f.score \mid f \ll f'\}$$

A dynamic programming algorithm based on this recurrence takes $O(mk^2)$ time to compute an optimal global chain, where $k$ is the number of fragments.

We can use the fact that we expect an optimal global chain to contain almost all of the fragments to reduce the expected time complexity to $O(k \log k + mk)$: We sort the fragments by increasing position in e.g. the first sequence and process them in this order. Moreover, we maintain the already processed fragments in an array sorted by score. For each fragment $f'$ we search for its predecessor of highest score as follows: We consider the already processed fragments (starting with the one with the highest score yet) and pick the first fragment which is actually a predecessor of $f'$. For almost all fragments, the number of other fragments checked will be one and the fragment will be inserted at the end of the array. Of course, this heuristic does not reduce the worst case time complexity, but it works very well in our context.

The fragments (multiMUMs) in an optimal global chain on the parse-level are the initial anchors of the multiple alignment of the $m$ sequences. Such an anchor (a multiMUM of the parses) cannot be extended on the parse-level, but in most cases it can be extended on the base-level. This is because the phrases corresponding to two different meta-symbols may share a common suffix (so that a left extension on the base-level may be possible) or prefix (ditto for a right extension). If we replace each meta-symbol in a multiMUM on the parse-level with its corresponding phrase from the dictionary, then we obtain a multiple exact match on the base-level. We simultaneously extend it base-by-base to the left (and right, respectively) in each of the $m$ sequences until a mismatch occurs. Since such a mismatch is most likely a SNP, we try to further extend it by the same procedure. This iterative extension ends when less than 10 bases

match exactly (simultaneously in all of the sequences) beyond one mismatch. This is illustrated in Step 3 of Figure 3: The anchors I, II, and IV are extended on the base-level (anchor I is extended twice to the left). After this extension step, we have the final anchors (the backbone) of the overall multiple alignment.

## 6 GENERATING ALIGNMENTS OF THE GAPS

After the computation of the anchors, we need to fill the gaps between them. (Note that this can trivially be parallelized because the alignments of two gaps are independent of each other.) For small gaps, we can simply use another multiple-sequence aligner (we use FMAlign2 (Zhang et al., 2024) and MAFFT (Nakamura et al., 2018)). However, when there are large gaps, e.g. because the sequences are incomplete, those other programs take very long to run.

For this reason, we try to fill the gaps using *partial* `multiMUMs`. A partial `multiMUM` is a `multiMUM` of a non-empty subset $\mathcal{S}$ of the sequences. For convenience, we denote it by $(\ell, p_1, \ldots, p_m)$, where $p_i$ is the position of the `multiMUM` in sequence $i$ if this sequence is in $\mathcal{S}$ and $p_i = \perp$ otherwise (where $\perp$ is a special value indicating absence). We say the partial `multiMUM` *occurs* in sequence $i$ if and only if $p_i \neq \perp$, and for all $i \in \{1, \ldots, m\}$ with $p_i \neq \perp$ it is required that the string $S^i[p_i..p_i + \ell - 1]$ occurs exactly once in $S^i$. [3]

In what follows, we describe our process of completing the alignment (Phase 3 in the Introduction) in the six steps that are visualized in Figure 4. The Steps 1–5 correspond to Phase 3a and Step 6 equals Phase 3b.

**Step 1: Find Partial `multiMUMs`.** The first step is to find partial `multiMUMs`. By definition, this can be achieved using a trivial modification (replacing $rb - lb + 1 = m$ with $rb - lb + 1 \leq m$ in Lemma 2) of the algorithm for finding `multiMUMs`. If the gap is sufficiently large, we work on the parse-level, otherwise we work on the base-level. If a previous attempt on the parse-level of this gap did not yield any new anchors, we also switch to the base-level. For

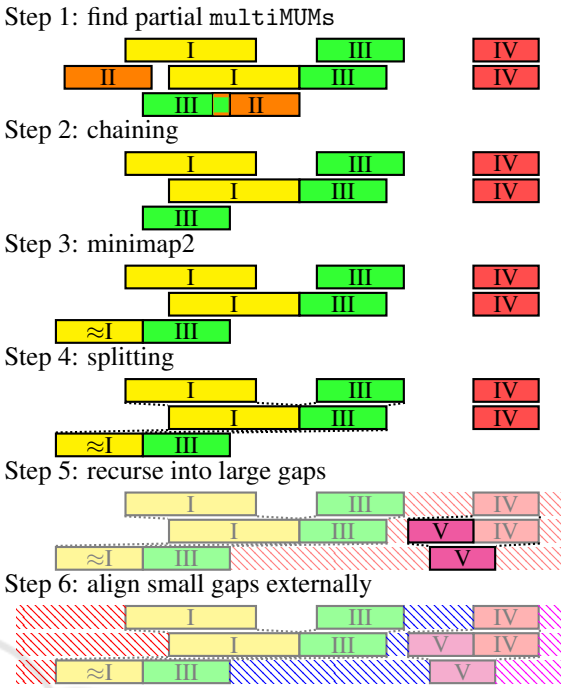


Figure 4: Schematic filling of a large gap. Note that it is possible to find `multiMUMs` that span all sequences (such as III) which were not discovered when considering all sequences. This can happen when the same string also appears somewhere outside the large gap, or because we compute `multiMUMs` on the base-level instead of on the parse-level.

algorithmic reasons (see below), we require each partial `multiMUM` to occur in more than half of the sequences.

**Step 2: Chain Partial `multiMUMs`.** As in the computation of the backbone, we chain the fragments (partial `multiMUMs`). However, the weight of a fragment must now also incorporate the number of sequences the fragment occurs in. As new weight, we use the length of the fragment multiplied by the number of sequences it occurs in (i.e. the total number of bases it covers, as does FMAlign2 (Zhang et al., 2024)). Unfortunately, chaining partial fragments appears to be fundamentally more difficult than chaining fragments occurring in all sequences. This is due to the fact that the $\ll$-relation on partial fragments is not transitive anymore.[4] To circumvent this issue, we use a heuristic. Specifically, we select $k$ sequences and then chain only those partial `multiMUMs` occurring in all of these $k$ sequences. The selected sequences are those with the most bases covered by the found partial `multiMUMs`. We do this for all $k \in \{\lfloor \frac{m}{2} \rfloor + 1, \ldots, m\}$,

[3] Note that this definition does not require the converse to be true (i.e. if the string $S^i[p_i..p_i + \ell - 1]$ occurs exactly once in sequence $i$, the partial `multiMUM` must not necessarily occur in sequence $i$). This is because our algorithm for chaining partial `multiMUMs` may discard the occurrence of a partial `multiMUM` in a sequence in order to increase the overall score of the chain (although this is unlikely).

[4] All (polynomial-time) chaining algorithms known to us rely on this transitivity.

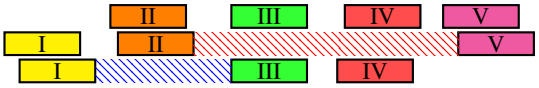


Figure 5: Since we require that each anchor occurs in more than half of the sequences, they can be linearly ordered (I $\ll$ II $\ll$ III $\ll$ IV $\ll$ V). Consider the gap in the second sequence between II and V (marked red). The anchors III and IV are between II and V in the linear order, thus we try to map them (the string they represent) to this gap using minimap2. Similarly, we would try to map II to the gap between I and III in the third sequence (marked blue), I to the gap before II in the first sequence, and V to the third sequence after IV.

and then choose the resulting chain with the most bases covered. The restriction $k > \frac{m}{2}$ is required for every pair of partial `multiMUMs` to be either incompatible (i.e. overlapping or not colinear) or comparable with $\ll$, and thus ensures that the anchors can be totally ordered.

The partial `multiMUMs` in the resulting 'sub-chain' may have had occurrences in the $m-k$ other sequences, which should of course not be discarded. Thus, we add them to the chain with a simple variant of the well-known algorithm for the heaviest increasing subsequence (HIS) (Jacobson and Vo, 1992). Since the order of the partial `multiMUMs` in the sub-chain is fixed, this can be done separately for each sequence.

**Step 3: Extend Partial Fragments with Minimap2.** After computing such a sub-chain, we try to extend each resulting partial fragment to the sequences it does not occur in using the long read mapper minimap2 (Li, 2021). This is illustrated in Figure 5.

Recall that the fragments are linearly ordered because each occurs in more than half the sequences. Let those fragments be $f_1 \ll \cdots \ll f_k$. We consider each gap in each sequence separately. For such a gap, we determine the two fragments $f_i$ and $f_j$ ($1 \leq i < j \leq k$) that bound it and try to map the fragments between them ($\{f_{i+1}, \ldots, f_{j-1}\}$) to this gap using minimap2. The matches reported by minimap2 are then chained using the same HIS variant as above.

Now the fragments in the sub-chain are not changed anymore and are considered anchors.

**Steps 4–6: Complete Alignment.** After adding the new anchors, we again try to split the large gap into smaller gaps, which are independent of each other.

If a resulting gap is sufficiently small, we use FMAlign2 to align it, otherwise we recurse. It may be the case that, after splitting, a gap still contains some anchors. (This happens when anchors occur in too few sequences and do not lead to a split.) These anchors are kept, and we ensure that chains in lower recursion levels are consistent with them.

If there is a sequence in a large gap that cannot be aligned to the other sequences via (partial) `multiMUMs` or minimap2, this gap cannot be split. However, since we can assume that there is no match anyway (because neither minimap2 nor we could find one), we remove this sequence from this gap before recursing and 'align' it immediately after the gap. This is only done when there are anchors exceeding a given length that span the other sequences.

## 7 EXPERIMENTAL RESULTS

We implemented our algorithm in an experimental tool, called PANAMA, in `C++`. The source code is publicly available.[5] For the construction of enhanced suffix arrays we used the `SDSL-lite` library (Gog et al., 2014).

We evaluated our implementation on two datasets, namely data from the 1000 Genomes Project[6] (56 GiB) and data from the draft human pangenome reference[7] (21 GiB) (Liao et al., 2023). The generated alignments are publicly available.[8]

The first dataset stems from (Boucher et al., 2021) and contains 1000 human haplotypes of chromosome 19. The sequences within this set all have a length of about 59 million base pairs. It can be viewed as a set of complete telomere-to-telomere assemblies (there is no missing data).

The second dataset contains the phased, diploid assemblies of 47 genetically diverse humans. In order to determine the ordering and orientation of the contigs of the assemblies, we utilized the mapping software gedmap (Büchler et al., 2023) as a long read mapper. For each phased assembly $A$, we split each contig $c_A$ of $A$ into 500bp long reads and mapped these to the reference genome hs37d5 including the variants from the 1000 genome project (The 1000 Genomes Project Consortium, 2015). The resulting alignments were chained and the best chain was used to map $c_A$ to the reference. In most cases, the best chain distinctly revealed the position and orientation of $c_A$ relative to the reference. When a contig $c_A$ could not be placed, it was excluded from further process-

[5] https://gitlab.com/qwerzuiop/panama

[6] http://dolomit.cs.tu-dortmund.de/tudocomp/chr19.1000.fa.xz archived at https://www.uni-ulm.de/in/theo/research/seqana/panama

[7] https://s3-us-west-2.amazonaws.com/human-pangenomics/index.html?prefix=working

[8] https://www.uni-ulm.de/in/theo/research/seqana/panama

Table 1: Wall clock time in minutes of the individual phases for our two test sets and two PFP-moduli, and the maximum RAM usage in GiB. The numbering of the phases refers to the overview in the Introduction.

| $\text{dataset}_{\text{modulus}}$ | $c19_{100}$ | $c19_{20}$ | $c1_{100}$ | $c1_{20}$ |
|---|---|---|---|---|
| 1: PFP | 34 | 47 | 15 | 20 |
| 2: find backbone | 28 | 125 | 9 | 35 |
| 3: align gaps | 22 | 23 | 40 | 40 |
| total | 98 | 195 | 64 | 95 |
| RAM | 64 | 69 | 51 | 111 |

ing. We selected all contigs of *A* that were aligned to chromosome 1 (the largest human chromosome) and sorted them by their position. If necessary, we replaced them with their reverse complement. The concatenation of these ordered contigs (separated by special symbols) was used as the (partial) DNA sequence of chromosome 1 of assembly *A*. In our experiment, we included all sequences (obtained in this way) that covered at least 80% of chromosome 1, which was the case for 92 out of the 94 assemblies. It is thus not surprising that the lengths of the sequences in this set range from 203 to 250 million base pairs. This means that we allowed up to 20% missing data per sequence, which is an additional challenge when generating the alignment.

For filling the small gaps (at most $10^7$ bases), we used FMAlign2 (Zhang et al., 2024) with HAlign3 (Tang et al., 2022) as backend, and fell back to MAFFT (Nakamura et al., 2018) in case FMAlign2 produced no output.[9] We accepted a match from minimap2 when the reported `MAPQ` ('mapping quality') value of the mapping was at least 30.[10] Since the modulus used in PFP for determining trigger strings influences the expected length of the phrases, we evaluated our program PANAMA with different moduli (20 and 100). The window size for the KR-hashing was always 10, because it seems to not significantly affect the properties of the parsing as long as it is sufficiently large (Boucher et al., 2019). Note that a window size of 10 is also the default used in (Boucher et al., 2019). The modulus for PFP in large gaps was always 20.

All experiments were conducted on a Linux-6.5.0 machine with an AMD EPYC 7742 (64 cores, 128 threads) processor and 256GB of RAM. We compiled PANAMA with GCC 11.4.0 with optimization flags `-O3 -funroll-loops -march=native -DNDEBUG`. The reported wall clock time was measured with the utility GNU Time. The reported RAM usage was calculated by measuring the maximum resident system RAM usage during execution and subtracting the resident system RAM usage before/after execution.[11] The results of our experiments can be seen in Tables 1 and 2. By far the most memory hungry phase is Phase 3, which invokes external aligners in parallel (the external aligners are responsible for most of the maximum RAM usage here). Therefore, the memory usage of PANAMA could greatly be reduced by reducing the number of concurrent threads.

There is no data missing in the chromosome 19 dataset, thus the gaps in the backbone (i.e. after extension 2d) are small. Tables 1 and 2 show that PANAMA can generate a high quality multiple alignment of 1000 haplotypes of chromosome 19 within hours. (Of course, such a high quality alignment is possible only if the input sequences are very similar, which is the case here.)

In contrast, on the chromosome 1 dataset, the number of bases per sequence varies greatly because we had to use contigs instead of whole chromosomes. This variance leads to large gaps in the backbone. In particular, there is one large gap for which PANAMA could not produce a reasonable alignment. This gap is located around the centromere (Logsdon et al., 2024), starting at around base $1.03 \cdot 10^8$ and ending at around base $1.36 \cdot 10^8$.[12] The centromere is known to be particularly difficult to align as it is highly repetitive (Logsdon et al., 2024). Since PANAMA was unable to generate a reasonable alignment of the centromeric region, the coverage is 'only' 81%, see Table 2.

Note that the suffix array construction algorithms in the `SDSL-lite` library (`DivSufSort` and `qsufsort`) are not the fastest ones that exist (see e.g. (Olbrich et al., 2024) for a recent comparison) and are not parallelized. Moreover, the library is generally focused on low memory usage instead of runtime. Therefore, we conjecture that the performance (especially of Phase 2a 'compute GSA') of our implementation can be significantly improved.

We also ran FMAlign2 (with HAlign3 as backend), HAlign3, and MAFFT v7.490 (all with 128

[9] FMAlign2 crashes on some inputs. This affects mostly simple cases, e.g. where all sequences are prefixes of the same string.

[10] This corresponds to probability of the mapping being wrong of at most 0.01%. Note that after accepting matches, we still chain them.

[11] Since FMAlign2 and PANAMA execute other external programs, measuring RAM usage with e.g. GNU Time is not sufficient.

[12] Since the reference genome is not part of our dataset, the stated positions are estimated as the number of bases preceding the gap. The start position of the gap varies by less than $10^6$ bases, but the number of bases per sequence in this gap ranges from $1.2 \cdot 10^7$ to $5.7 \cdot 10^7$. The reported end position is the start position plus the median of the number of bases in the sequences in this gap.

Table 2: Evaluation of the generated alignments: The number of anchors equals the number of `multiMUMs` in the chain computed in Phase 2c. The number of extensions equals the number of left or right extension of anchors beyond a mismatching column made in Phase 2d. This can also be seen as the number of potentially detected SNPs at this phase. Percent coverage is the number of bases that are included in our alignment relative to the overall number of bases. Percent identity is the number of 100% identity columns (i.e. columns without gaps in which all bases are identical) relative to the alignment length. The centromeric region of chromosome 1 is included in all measurements but excluded from the computation of the identity.

| $\text{dataset}_{\text{modulus}}$ | $\text{c19}_{100}$ | $\text{c19}_{20}$ | $\text{c1}_{100}$ | $\text{c1}_{20}$ |
|---|---|---|---|---|
| # anchors | 88 788 | 305 480 | 336 206 | 640 794 |
| # extensions | 335 644 | 509 986 | 661 532 | 712 606 |
| coverage in %: | | | | |
| after 2c (chain) | 19.01 | 49.30 | 23.28 | 49.27 |
| after 2d (extend) | 65.84 | 92.57 | 63.98 | 70.32 |
| final alignment | 100 | 100 | 81.38 | 81.38 |
| identity in %: | 97.93 | 98.15 | 75.93 | 75.90 |

Table 3: Results of FMAlign2, HAlign3, and PANAMA (with PFP modulus 100) on a few haplotypes of chromosome 19. The given time is the wall clock time.
(a) FMAlign2 running on small sets of chromosome 19 haplotypes.

| #haplotypes | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| time (min) | 36 | 78 | 123 | 167 | 216 | 270 |
| RAM (GiB) | 19 | 27 | 35 | 43 | 51 | 59 |

(b) HAlign3 running on small sets of chromosome 19 haplotypes.

| #haplotypes | 2 | 4 | 16 | 24 | 26 |
|---|---|---|---|---|---|
| time (min) | 0.6 | 0.7 | 1.0 | 1.6 | 2.4 |
| RAM (GiB) | 10.0 | 12.2 | 26.5 | 32.2 | 32.5 |

(c) PANAMA running on small sets of chromosome 19 haplotypes.

| #haplotypes | 2 | 4 | 16 | 24 | 26 |
|---|---|---|---|---|---|
| time (min) | 0.2 | 0.4 | 1.2 | 1.7 | 1.8 |
| RAM (GiB) | 0.4 | 0.5 | 0.8 | 1.0 | 1.2 |

threads and otherwise default options) on sets of chromosome 19 haplotypes. The results are shown in Table 3. FMAlign2 is able to compute the alignment of 3 to 6 haplotypes in the time PANAMA requires for all 1000 (depending on the modulus used), but is not suitable for the whole chromosome 19 dataset. On these datasets, HAlign3 is much faster than FMAlign2 but still slower than PANAMA. HAlign3 crashed on cases with more than 26 haplotypes with a "`NullPointerException`". MAFFT already crashed on two chromosome 19 haplotypes with a "`TOO MANY SEGMENTS`" error.

# 8 CONCLUSION AND FUTURE WORK

The experiments showed that our tool PANAMA can very efficiently generate a pangenomic-scale multiple alignment of assembled genomes of the same species, provided that structural variants seldomly occur. It could be argued that in the presence of large-scale chromosomal rearrangements, a (colinear) multiple alignment of the chromosomes does not make sense at all. In that case, a pangenome graph is a better representation of the genomes (Baaijens et al., 2022).

We plan to extend our work in such a way that it can deal with large-scale structural variants. Our approach can be modified to address this very important topic as follows: instead of computing a multiple alignment, it should be possible to use the syntenic regions as the backbone of a pangenome graph. In the construction of the full pangenome graph, transpositions and inversions can be detected. Moreover, programs for pangenome graph generation can be used to deal with the remaining (relatively small) gaps between anchors.

# REFERENCES

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86.

Baaijens, J. et al. (2022). Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, 21(1):81–108.

Boucher, C. et al. (2019). Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1).

Boucher, C. et al. (2021). PHONI: Streamed matching

statistics with multi-genome references. In *Data Compression Conference (DCC)*, pages 193–202. IEEE.

Büchler, T., Olbrich, J., and Ohlebusch, E. (2023). Efficient short read mapping to a pangenome that is represented by a graph of ED strings. *Bioinformatics*, 39(5):btad320.

Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center.

Chain, P. et al. (2003). An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105–123.

Cormen, T., Leiserson, C., and Rivest, R. (1990). *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Gog, S. et al. (2014). From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms (SEA)*, pages 326–337. Springer.

Höhl, M., Kurtz, S., and Ohlebusch, E. (2002). Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320.

Jacobson, G. and Vo, K. (1992). Heaviest increasing/common subsequence problems. In *Combinatorial Pattern Matching (CPM)*, pages 52–66. Springer.

Kasai, T. et al. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, pages 181–192. Springer.

Li, H. (2021). New strategies to improve minimap2 alignment accuracy. *Bioinformatics*, 37(23):4572–4574.

Liao, W. et al. (2023). A draft human pangenome reference. *Nature*, 617(7960):312–324.

Logsdon, G. et al. (2024). The variation and evolution of complete human centromeres. *Nature*, 629:136–145.

Louza, F., Gog, S., and Telles, P. (2017). Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39.

Nakamura, T. et al. (2018). Parallelization of MAFFT for large-scale multiple sequence alignments. *Bioinformatics*, 34(14):2490–2492.

Ohlebusch, E. and Kurtz, S. (2008). Space efficient computation of rare maximal exact matches between multiple sequences. *Journal of Computational Biology*, 15(4):357–377.

Olbrich, J., Ohlebusch, E., and Büchler, T. (2024). Generic non-recursive suffix array construction. *ACM Transactions on Algorithms*, 20(2):18.

Porubsky, D. et al. (2021). Fully phased human genome assembly without parental data using single-cell strand sequencing and long reads. *Nature Biotechnology*, 39(3):302–308.

Puglisi, S., Smyth, W., and Turpin, A. (2007). A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4.

Tang, F. et al. (2022). HAlign 3: fast multiple alignment of ultra-large numbers of similar DNA/RNA sequences. *Molecular Biology and Evolution*, 39(8):msac166.

Tettelin, H. et al. (2005). 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.

The 1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, 526(7571):68–74.

Treangen, T. et al. (2014). The Harvest suite for rapid core-genome alignment and visualization of thousands of intraspecific microbial genomes. *Genome Biology*, 15(11):524.

Zhang, P. et al. (2024). FMAlign2: a novel fast multiple nucleotide sequence alignment method for ultralong datasets. *Bioinformatics*, 40(1):btae014.

# APPENDIX

## A Enumeration of LCP-Intervals

(Kasai et al., 2001) presented a linear time algorithm to simulate the bottom-up traversal of a suffix tree with a suffix array and its LCP-array (which, given the suffix array, can be constructed in linear time). The following algorithm is a slight modification of their algorithm TraverseWithArray, cf. (Abouelhoda et al., 2004). It computes all lcp-intervals of the LCP-array with the help of a stack. The elements on the stack are lcp-intervals represented by tuples $\langle lcp, lb, rb \rangle$, where $lcp$ is the lcp-value of the interval, $lb$ is its left boundary, and $rb$ is its right boundary. In Algorithm 1, push (pushes an element onto the stack) and pop (pops an element from the stack and returns that element) are the usual stack operations, while top provides a pointer to the topmost element of the stack. Furthermore, $\perp$ stands for an undefined value. We assume that array indexing starts at 1 and that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n+1]$.

**Function** `Enumerate(`LCP`)`**:**
  $push(\langle 0, 1, \perp \rangle)$;
  **for** $k = 2 \to n+1$ **do**
    $lb \leftarrow k-1$;
    **while** $\mathsf{LCP}[k] < top().lcp$ **do**
      $top().rb \leftarrow k-1$;
      $interval \leftarrow pop()$;
      $report(interval)$;
      $lb \leftarrow interval.lb$;
    **end**
    **if** $\mathsf{LCP}[k] > top().lcp$ **then**
      $push(\langle \mathsf{LCP}[k], lb, \perp \rangle)$;
    **end**
  **end**

Algorithm 1: Given the LCP-array of a string of length $n$, this algorithm enumerates all lcp-intervals.