HAL
open science

Linking indexing data structures to de Bruijn graphs: Construction and update

Bastien Cazaux, Thierry Lecroq, Eric Rivals

To cite this version:

Bastien Cazaux, Thierry Lecroq, Eric Rivals. Linking indexing data structures to de Bruijn graphs: Construction and update. Journal of Computer and System Sciences, 2019, 104, pp.165-183. 10.1016/j.jcss.2016.06.008 . lirmm-01617207

HAL Id: lirmm-01617207

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01617207v1

Submitted on 16 Oct 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ARTICLE IN PRESS YJCSS:2997

Journal of Computer and System Sciences ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Linking indexing data structures to de Bruijn graphs: Construction and update

Bastien Cazaux [a,b], Thierry Lecroq [c], Eric Rivals [a,b,*]

[a] LIRMM, CNRS and Université de Montpellier, 161 rue Ada, 34095 Montpellier Cedex 5, France
[b] Institut Biologie Computationnelle, CNRS and Université de Montpellier, 860 rue Saint Priest, 34095 Montpellier Cedex 5, France
[c] Normandie Univ. & UNIROUEN, UNIHAVRE, INSA Rouen, LITIS, 76000 Rouen France

ARTICLE INFO

Article history:
Received 25 June 2015
Received in revised form 26 May 2016
Accepted 27 June 2016
Available online xxxx

Keywords:
Index
Data structure
Suffix tree
Suffix array
Dynamic update
Overlap
Contracted de Bruijn graph
Assembly
Algorithms
Bioinformatics

ABSTRACT

DNA sequencing technologies have tremendously increased their throughput, and hence complicated DNA assembly. Numerous assembly programs use de Bruijn graphs (dBG) built from short reads to merge these into contigs, which represent putative DNA segments. In a dBG of order k, nodes are substrings of length k of reads (or k-mers), while arcs are their $k+1$-mers. As analysing reads often require to index all their substrings, it is interesting to exhibit algorithms that directly build a dBG from a pre-existing index, and especially a contracted dBG, where non-branching paths are condensed into single nodes. Here, we exhibit linear time algorithms for constructing the full or contracted dBGs from suffix trees, suffix arrays, and truncated suffix trees. With the latter the construction uses a space that is linear in the size of the dBG. Finally, we also provide algorithms to dynamically update the order of the graph without reconstructing it.

© 2016 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

In life sciences, determining the sequence of bio-molecules is an essential step towards the understanding of their functions and interactions within an organism. Powerful sequencing technologies allow to get huge quantities of short sequencing reads that need to be assembled to infer the complete target sequence. These constraints favour the use of a version of the de Bruijn Graph (dBG) dedicated to genome assembly – a version which differs from the combinatorial structure invented by N.G. de Bruijn [1]. Given a set $S = \{s_1, \ldots, s_n\}$ of n reads and an integer k, an assembly de Bruijn Graph, or for short simply de Bruijn Graph, stores each k-mer (k-long substring) occurring in the reads as nodes and has an arc joining two k-mers if they appear as successive (and hence overlapping) k-mers in at least one read.

The dBG is then traversed to extract long paths, which will form the *contigs*, *i.e.*, the sequence of sub-regions of the molecule. In non-repetitive regions, the layout of the reads dictates a simple path of k-mers without bifurcations. Any simple path between an in-branching node and the next out-branching node, can then be contracted into a single arc without loosing any information on the graph structure. The sequences of such simple paths are called *unitigs* (the contraction from unique and contigs). The version of the dBG where each such "non-branching" path is condensed into a single arc is termed the Contracted dBG (CdBG).

* Corresponding author at: LIRMM, CNRS and Université de Montpellier, 161 rue Ada, 34095 Montpellier Cedex 5, France.
E-mail addresses: bastien.cazaux@lirmm.fr (B. Cazaux), thierry.lecroq@univ-rouen.fr (T. Lecroq), rivals@lirmm.fr (E. Rivals).

http://dx.doi.org/10.1016/j.jcss.2016.06.008
0022-0000/© 2016 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Please cite this article in press as: B. Cazaux et al., Linking indexing data structures to de Bruijn graphs: Construction and update, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.06.008

Sequencing technologies of the second generation can yield hundreds of millions of reads. Compared to the overlap graph or to the string graph, which were used with previous technologies, the dBG has a number of nodes that is not proportional to the number of reads: it depends on a user controlled parameter k, termed the *order* of the dBG. Its memory usage can be fine tuned through this parameter.

In bioinformatics, dBGs are heavily exploited for genome assembly [2], but for other purposes as well. Actually, some programs mine the dBG to seek graph patterns representing mutations, large insertions/deletions, or chromosomal rearrangements [3]. Others use it to correct sequencing errors in long reads [4].

The de Bruijn Graph is usually built directly from the set of reads, which is time and space consuming. Several compact data structures for storing dBGs have been developed [5,6] including probabilistic ones [7]. The emphasis is placed on the practical space needed to store the dBGs in memory. Moreover, some recent assembly algorithms put forward the advantage of using for the same input, multiple dBGs with increasing orders [8], thereby emphasising the need for dynamically updating the dBG. In all cases, the construction algorithms need to scan through the whole set of reads.

Several genome assembly programs used hash tables to store the k-mer of the reads and allow navigating through the arcs of the dBG, but these solutions suffer from several limitations regarding *e.g.* functionalities and flexibility. With hash functions, it is often not possible to add extra information to the nodes, like for instance the number of times a k-mer is observed in the read set, which is used as a confidence measure. Hash tables make it difficult to compute the contracted dBG or to change the value of k. The main advantage of sophisticated hash functions is their memory footprint. For instance Minia [9] offers a very space efficient storage to handle the dBG based on cascading Bloom filters, which are a type of hash functions. This hash table based solution was used for long read error correction and also proves efficient in that context [4].

In studies involving the analysis of sequencing reads, distinct tasks require to index either all substrings, or the k-mers of the reads. For instance, fast dBG assembly programs first count the k-mers before building the dBG to estimate the memory needed [7]. Another example: some error correction software build a suffix tree of all short reads to correct them [10]. Hence, before the assembly starts, the read set has already been scanned through and indexed. It can thus be efficient to enable the construction of the dBG for the subsequent assembly, directly from the index rather than from scratch. For these reasons, we set out to find algorithms that transform usual indexes into a dBG or a contracted dBG. It is also of theoretical interest to build bridges between well studied indexes and this graph on words. Despite recent results [11,12], formal methods for constructing dBGs from suffix trees are an open question. In comparison, Simpson and Durbin have proposed an algorithm to build the String Graph from a FM-index [13].

Here, we present algorithms to build directly the CdBG from a Generalised Suffix Tree or from a Generalised Suffix Array of the reads [14–17]. These algorithms take space and time that are linear in the input size. These well-known data structures index all substrings of the reads, and not only their k-mers. This results in one drawback and in one advantage.

The drawback is their space occupancy. We will then consider an indexing data structure that reduces the set of indexed substrings: the truncated suffix tree [18,19]. We introduce the reduced truncated suffix tree (TST) and then show how to construct with this index both the dBG and CdBG in time and space that are linear in the size of the final dBG, rather than in the cumulated length of the reads. By size of the dBG we mean the sum of number of nodes, plus the number of arcs. This algorithm achieves an optimal time and space complexity.

The advantage is the counterpart: as substrings of all lengths are indexed, it allows to update the order of the graph, that is to change dynamically the value of k without reconstructing the dBG. Finally, we provide efficient algorithms for increasing or decreasing the value of k. Of course, if one uses the truncated suffix tree instead of the full suffix tree, only some updates remain possible. Our results nevertheless remain applicable to the truncated suffix tree, where the order can be dynamically decreased.

This article includes results that appeared in [20,21].

1.1. Indexing data structures

Suffix trees are well-known indexing data structures that enable to store and retrieve all the factors of a given string. The suffix tree of a string y of length s can be build in time and space in $O(s)$ on a constant size alphabet [14,22]. Then, it is possible to check if a pattern x of length m is a factor of a string y of S in time $O(m)$. Counting the number of occurrences of x in y can also be done in time $O(m)$ while enumerating the positions where x occurs in y can be performed in time $O(m + occ)$, where occ denotes the number of occurrences of x in y. Suffix trees can be adapted to a finite set of strings and are then called Generalised Suffix Trees (GSTs). Thus, given a set S of n strings of total length $\|S\|$ on a constant size alphabet, the generalised suffix tree for S can be build in time and space $O(\|S\|)$. For a detailed exposition of properties of suffix trees we refer the reader to [17]. Suffix trees have been widely studied and used in a large number of applications (see [15] and [17]). In practice, they consume too much space and are often replaced by the more economical suffix arrays [16], which have the same properties [23].

When one is only interested in factors of a given length, truncated suffix trees only store the factors of length up to a given constant k of a given string. They can also be build in linear time and space [18]. In practice, truncated suffix trees save a lot of nodes compared to suffix trees.

	1	2	3	4	5	6	7
s_1	b	a	c	b	a	b	
s_2	c	b	a	b	c	a	a
s_3	b	c	a	a	c	b	
s_4	c	b	a	a	c		
s_5	b	b	a	c	b	a	a

Fig. 1. $S := \{bacbab, cbabcaa, bcaacb, cbaac, bbacbaa\}$ is a set of words. Therefore, we have $Support(ba) = \{(1,1),(1,4),(2,2),(4,2),(5,2),(5,5)\}$, $RC(ba) = \{\varepsilon, c, cb, cba, cbab, b, bc, bca, bcaa, a, ac, cbaa\}$, $LC(ba) = \{\varepsilon, c, ac, bac, b, bbac\}$ and $d(ba) = 0$. One has $RC(ba) \cap \Sigma = \{a, b, c\}$. Thus, the word ba is not right extensible in S (see Definition 2).

2. Definitions of de Bruijn graphs

2.1. Notation about strings

Here we introduce a notation and basic definitions.

An *alphabet* Σ is a finite set of *letters*. A finite sequence of elements of Σ is called a *word* or a *string*. The set of all words over Σ is denoted by $\Sigma^\star$, and ε denotes the empty word. For a word x, $|x|$ denotes the *length* of x. Given two words x and y, we denote by $x \cdot y$ or simply xy the *concatenation* of x and y. For every $1 \le i \le j \le |x|$, $x[i]$ denotes the i-th letter of x, and $x[i\,..\,j]$ denotes the *substring* or *factor* $x[i]x[i+1]\ldots x[j]$. Let k be a positive integer. If $|x| \ge k$, $first_k(x)$ is the *prefix* of length k of x and $last_k(x)$ is the *suffix* of length k of x. Then a substring of length k of x is called a k-mer of x. For i such that $1 \le i \le |x| - k + 1$, $(x)_{k,i}$ is the k-mer of x starting in position i, *i.e.*, $(x)_{k,i} = x[i\,..\,i+k-1]$. Thus we have $first_k(x) = (x)_{k,1}$ and $last_k(x) = (x)_{k,|x|-k+1}$. We denote by $\sharp(\Lambda)$ the cardinality of any finite set Λ.

Let $S = \{s_1, \ldots, s_n\}$ be a finite set of words. It is our running instance for all the following. Let us denote the sum of the lengths of the input strings by

$$\|S\| := \sum_{s_i \in S} |s_i|$$

We denote by

- $F(S)$ the set of factors of words of S, *i.e.*, $F(S) = \{w \in \Sigma^\star \mid \exists u, v \in \Sigma^\star, 1 \le i \le n, s_i = uwv\}$.
- $F_k(S)$ the set of factors of length k of S where k is a positive integer, *i.e.*, $F_k(S) = F(S) \cap \Sigma^k$.
- $Suff_k(S)$ is the set of suffixes of length k of words of S.

2.2. Classical definition of de Bruijn graph

All definitions below refer to the set S; however, as S is clear from the context, we simply omit the "in S" in the notation.

For a word w of $F(S)$,

- $Support(w)$ is the set of pairs (i, j), where w is the substring $(s_i)_{|w|,j}$. $Support(w)$ is called the support of w in S.
- $RC(w)$ (resp. $LC(w)$) is the set of *right context* (resp. *left context*) of the word w in S, *i.e.*, the set of words w' such that $ww' \in F(S)$ (resp. $w'w \in F(S)$).
- $\lceil w \rceil$ is the word ww' where w' is the longest word of $RC(w)$ such that $Support(w) = Support(ww')$. In other words, such that w and ww' have exactly the same support in S.
- $\lfloor w \rfloor$ is the word w' where w' is the longest prefix of w such that $Support(w') \neq Support(w)$.
- $d(w) := |\lceil w \rceil| - |w|$.

In other words, $\lceil w \rceil$ is the longest extension of w having the same support as w in S, while $\lfloor w \rfloor$ is the shortest reduction of w with a support different from that of w in S. These definitions are illustrated in a running example presented in Fig. 1.

We give the definition of a de Bruijn graph for assembly (dBG for short), which differs from the original definition of a complete graph over all possible words of length k stated by de Bruijn [1].

Definition 1. Let k be a positive integer. The *de Bruijn graph* of order k for S, denoted by DBG_k^+, is a directed graph, $DBG_k^+ := (V_k^+, E_k^+)$, whose vertices are the k-mers of words of S and where an arc links u to v if and only if u and v are two successive k-mers of a word of S, *i.e.*:

$$V_k^+ := F_k(S)$$

$$E_k^+ := \{(u, v) \in {V_k^+}^2 \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } v[k] \in RC(u)\}.$$

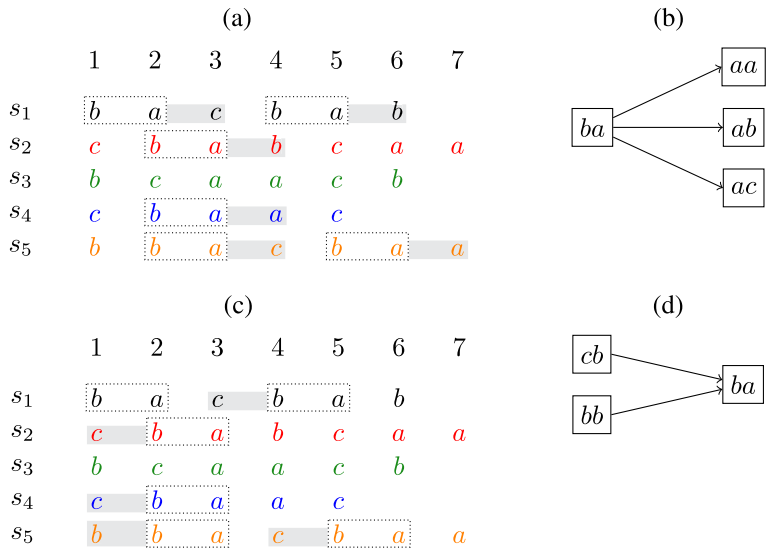

Fig. 2. Examples of arcs from DBG_k^+. (a) shows letters in the right context of *ba*, and (b) the successors of node *ba* in DBG_2^+; one for each letter in $RC(w) \cap \Sigma$. (c) shows letters in the left context of *ba*, and (d) the predecessors of node *ba* in DBG_2^+; one for each letter in $LC(w) \cap \Sigma$.

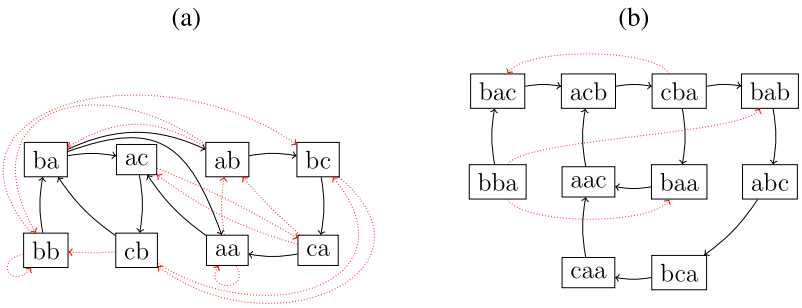

Fig. 3. With solid arcs only, the graphs correspond to DBG_2^+ (a) and DBG_3^+ (b) for our running example. With both solid and dotted arcs, they represent DBG_2^- (a) and DBG_3^- (b).

An equivalent definition of E_k^+ can be stated using the left instead of right context:

$$E_k^+ := \{(u, v) \in V_k^{+2} \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } u[1] \in LC(v)\}.$$

Examples of arcs are displayed on Fig. 2. The size of DBG_k^+ is denoted by and defined as $size(DBG_k^+) := \sharp(V_k^+) + \sharp(E_k^+)$. Note that another, simpler definition of the arcs in the de Bruijn graph coexists with that of Definition 1. There, an arc links u to v if and only if u overlaps v by $k-1$ symbols. This graph is denoted by $DBG_k^- = (V_k^-, E_k^-)$, where:

$$V_k^- := F_k(S)$$
$$E_k^- := \{(u, v) \in V_k^{-2} \mid last_{k-1}(u) = first_{k-1}(v)\}.$$

The arcs of E_k^- satisfy less constraints than those of E_k^+; hence, E_k^+ is a subset of E_k^-. Both definitions are illustrated on Fig. 3. Some assembly programs use DBG_k^- [9]. All the algorithmic results that we obtain for DBG_k^+ remain valid for DBG_k^-. In the sequel, we focus only on DBG_k^+.

Let us introduce now the notions of extensibility for a substring of S and that of a Contracted dBG (CdBG for short).

Definition 2 *(Extensibility).* Let w be a word of $F(S)$.

- w is *right extensible* in S if and only if $\sharp(RC(w) \cap \Sigma) = 1$.
- w is *left extensible* in S if and only if $\sharp(LC(w) \cap \Sigma) = 1$.

Let w be a word of $\Sigma^\star$. The word w is said to be a *unique* k'-mer of S if and only if $k' \geq k$ and for all $i \in [1..k'-k+1]$, $(w)_{k,i} \in F(S)$ and for all $j \in [1..k'-k]$, $(w)_{k,j}$ is right extensible and $(w)_{k,j+1}$ is left extensible.

Please cite this article in press as: B. Cazaux et al., Linking indexing data structures to de Bruijn graphs: Construction and update, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.06.008

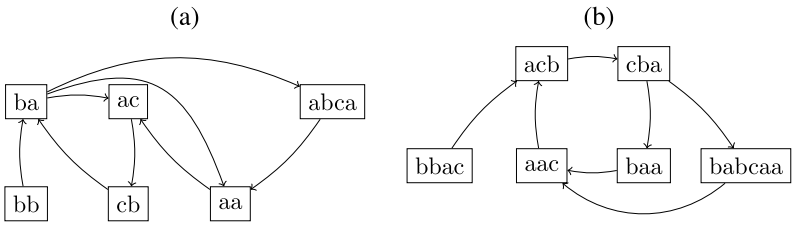

Fig. 4. The graphs correspond to CDBG_2^+ (a) and CDBG_3^+ (b) for our running example.

Definition 3. A *contracted de Bruijn graph* of order k, denoted by $\mathrm{CDBG}_k^+ = (V_{k,c}^+, E_{k,c}^+)$, is a directed graph where:

$$V_{k,c}^+ = \{w \in \Sigma^\star \mid w \text{ is a } k'\text{-mer unique maximal by substring and } k' \geq k\}$$
$$E_{k,c}^+ = \{(u, v) \in {V_{k,c}^+}^2 \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } v[k] \in RC(last_k(u))\}.$$

Examples of CDBG_k^+ are displayed on Fig. 4. Note that in the previous definition, an element w in $V_{k,c}^+$ does not necessarily belong to $F(S)$, since w may only exist as the substring of the agglomeration of two words of S. Thus, let w be a k'-mer unique maximal by substring with $k' \geq k$:

- $last_k(w)$ is not right extensible or $RC(last_k(w)) \cap \Sigma = \{a\}$ and $last_{k-1}(w) \cdot a$ is not left extensible,
- $first_k(w)$ is not left extensible or $LC(first_k(w)) \cap \Sigma = \{a\}$ and $a \cdot first_{k-1}(w)$ is not right extensible.

With this argument, we have both following propositions.

Proposition 1. *Let $(u, v) \in E_{k,c}^+$; $(last_k(u), first_k(v)) \in E_k^+$ and there exists $w \in V_k^+$ such that $(w, first_k(v)) \in E_k^+ \setminus \{(last_k(u), first_k(v))\}$ or $(last_k(u), w) \in E_k^+ \setminus \{(last_k(u), first_k(v))\}$.*

Proposition 2. *Let $(u, v) \in E_k^+$. If u is right extensible and v is left extensible, then there exists $w \in V_{k,c}^+$ such that $u \cdot v[k]$ is a substring of w. Otherwise, there exists $(u', v') \in E_{k,c}^+$ such that $u = last_k(u')$ and $v = first_k(v')$.*

According to Propositions 1 and 2, CDBG_k^+ is the graph DBG_k^+ where the arcs (u, v) are contracted if and only if u is right extensible and v is left extensible.

2.3. Constructive characterisation of the de Bruijn graph

Let k be a positive integer. We define the following three subsets of $F(S)$.

- $InitExact_k = \{w \in F(S) \mid |w| = k \text{ and } d(w) = 0\}$
- $Init_k = \{w \in F(S) \mid |w| \geq k \text{ and } d(first_k(w)) = |w| - k\}$
- $SubInit_k = InitExact_{k-1}$

A word of $InitExact_k$ is either only the suffix of some s_i or has at least two right extensions, while the first k-mer of a word in $Init_k \setminus InitExact_k$ has only one right extension.

Proposition 3. $InitExact_k = Init_k \cap \{w \in F(S) \mid |w| = k\}$.

Proof. Let $w \in InitExact_k$. In this case, we get $first_k(w) = w$ and $|w| - k = 0$. This means that $d(first_k(w)) = |w| - k$ and therefore $w \in Init_k$. □

For w an element of $Init_k$, $first_k(w)$ is a k-mer of S. Given two words w_1 and w_2 of $Init_k$, $first_k(w_1)$ and $first_k(w_2)$ are distinct k-mers of S. Furthermore for each k-mer w' of S, there exists a word w of $Init_k$ such that $first_k(w) = w'$. From this, we get the following proposition.

Proposition 4. *There exists a bijection between $Init_k$ and the set of the k-mers of S.*

According to Definition 1 and Proposition 4, each vertex of DBG_k^+ can be assimilated to a unique element of $Init_k$. As the vertices of DBG_k^- are identical to those of DBG_k^+, there exists also a bijection between $Init_k$ and the set of vertices of DBG_k^-.

To define the arcs between the words of $Init_k$, which correspond to arcs of DBG_k^+, we need the following proposition, which states that each single letter that is a right extension of w gives rise to a single arc.

Proposition 5. *For $w \in InitExact_k$ and $a \in \Sigma \cap RC(w)$, there exists a unique $w' \in Init_k$ such that $last_{k-1}(w)a$ is a prefix of w'.*

Proof. Let w be a word of $InitExact_k$ and a a letter of $RC(w)$. By definition of right context, $last_{k-1}(w)a \in F(S)$. As $|last_{k-1}(w)a| = k$, there exists w' such that $last_{k-1}(w)a$ is a prefix of w' and $|last_{k-1}(w)a| + d(last_{k-1}(w)a) = |w'|$. By definition of $Init_k$, $w' \in Init_k$. □

The set $Init_k$ represents the nodes of DBG_k^+. Let us now build the set of arcs that is isomorphic to E_k^+. Let w be a word of $Init_k$ and $\mathrm{Succ}_k(w)$ denote the set of successors of $first_k(w)$: $\mathrm{Succ}_k(w) := \{x \in Init_k \mid (first_k(w), first_k(x)) \in E_k^+\}$. We know that for each letter a in $RC(w)$, there exists an arc from $first_k(w)$ to $first_k(last_{|w|-1}(w)a)$ in DBG_k^+. We consider two cases depending on the length of w:

Case 1. $|w| = k$.
According to Proposition 3, $w \in InitExact_k$ and hence $last_{k-1}(w) \in SubInit_k$. Therefore, the outgoing arcs of w in DBG_k^+ are the arcs from w to w' satisfying the condition of Proposition 5. Then,

$$\mathrm{Succ}_k(w) = \bigcup_{a \in \Sigma \cap RC(w)} \lceil last_{k-1}(w)a \rceil.$$

Case 2. $|w| > k$.
As w is longer than k, it contains the next k-mer; hence $first_k(last_{|w|-1}(w)a) = first_k(last_{|w|-1}(w))$, and there exists a unique outgoing arc of w: that from w to $\lceil w[2..k] \rceil$. Indeed, by definition of $Init_k$, $\lceil w[2..k] \rceil \in Init_k$, and thus

$$\mathrm{Succ}_k(w) = \{\lceil w[2..k] \rceil\}.$$

Now, we can build integrally DBG_k^+ or more exactly an isomorphic graph of DBG_k^+.

Theorem 1. *With the sets $Init_k$, $InitExact_k$ and $SubInit_k$, we can build an isomorphic graph of DBG_k^+ in linear time in the size of these sets.*

For simplicity, from now on, we confound the graph we build with DBG_k^+.

2.4. Constructive characterisation of the contracted de Bruijn graph

To do the same with CDBG_k^+, initially we begin by explaining the algorithm that we use to build this graph and in the second time we need to characterise the concepts of right and left extensibility in terms of word properties.

Our algorithm to build $CDBG_k^+$. We present a generic algorithm to build incrementally CDBG_k^+. It is explained in terms of words, and does not depend on any indexing data structure. In following sections, we will use this generic algorithm and explain how it can be performed efficiently using a specified indexing structure.

The main algorithm (Algorithm 2) explores DBG_k^+ to find the nodes kept in CDBG_k^+ and set all single arcs that represent whole non-branching paths of DBG_k^+ that are properly contracted. The key point is to find all starting nodes of simple paths and explore these paths from them; the exploration is done by Algorithm 1.

A more detailed explanation. First, note that to build DBG_k^+ it suffices to know the set $\mathrm{Succ}_k(.)$ for each node. The algorithm below simulates a traversal of DBG_k^+ without building it, and stores only one node per unique maximal k'-mer of DBG_k^+. For such a k'-mer, say m, we choose to represent it by the node v such that $first_k(v)$ is a prefix of m. In DBG_k^+, m is represented by a simple (*i.e.*, non-branching) path and v is its first node. In the traversal algorithm, for a current starting node v_c in $Init_k$, we traverse the simple path until we arrive at a node u having several successors or such that its only successor is not left extensible (*i.e.*, has several predecessors). In other words, until we find u such that u is not right extensible or next(u) is not left extensible. In DBG_k^+, there exists a simple path between v_c and u, and this must build a single node in CDBG_k^+. To contract this path, we choose to keep v_c, and for any successor w of u, we insert an arc between u and w, as this arc cannot be contracted. Noting that w necessarily starts a chain (having at least a single node), if w is not yet in CDBG_k^+, we launch a new path exploration starting from w, one gets that $first_k(w)$ is the prefix of a node of CDBG_k^+, and thus w can appropriately represent the path. Now, if w already belongs to CDBG_k^+, the case is trickier. If v_f stores the first v_c called by the procedure, it may not be the starting node of a path, but be anywhere inside a path. Two cases arise. If v_f is considered during the while loop, then it is not at the start of a simple

Algorithm 1: BuildAuxCDBG(V, E, v_f, v_c).

Input : The partial contracted graph CDBG_k^+ as (V, E), two nodes v_f and v_c. v_f the initial starting node, and v_c the current starting node.
Output: The updated contracted graph (V', E'), which now contains all paths starting from v_c.

```
begin
    u := v_c; mark u
    // search the node ending the chain that goes through v_c
    while u is right extensible and next(u) is left extensible do
        if v_f = next(u) then
            update (v_f, i) by (v_c, i) for all (v_f, i) ∈ E
            return (V \ {v_f}, E)
        u := next(u); mark u
    // now explore the path starting in the successor of u
    for w ∈ Succ_k(u) do
        if w ∈ V then
            (V, E) := (V, E ∪ {(v_c, w)});
        else
            (V, E) := BuildAuxCDBG(V ∪ {w}, E ∪ {(v_c, w)}, v_f, w); ;        // explore from node w
    return (V, E)
```

Algorithm 2: BuildCDBG(S).

Input : A set of words S.
Output: CDBG_k^+ of S.

```
begin
    (V, E) = (∅, ∅)
    // search for any node v of DBG_k^+ without predecessors
    // and build CDBG_k^+ from v
    for v ∈ Init_k do
        if there exists no w such that v ∈ Succ_k(w) then
            (V, E) := (V, E) ⋃ BuildAuxCDBG(V ∪ {v}, E, v, v)
    // explore DBG_k^+ from any node not yet visited
    for v_c an unmarked node of Init_k do
        (V, E) := (V, E) ⋃ BuildAuxCDBG(V ∪ {v_c}, E, v_c, v_c)
    return (V, E)
```

path: hence we must update V by exchanging v_f with v_c and terminate the exploration. Otherwise, v_f is traversed during the for loop (as the value of w), then it is a successor of u and the beginning of a simple path: we just add an arc linking v_c to w and stop. Finally, if w already belongs to V but $w \neq v_f$, we also add an arc linking v_c to w and stop.

The process performed by Algorithm 1 augments the partial graph CDBG_k^+ restrained to the nodes visited when exploring the path starting from v_c. It suffices now to ensure that all arcs of DBG_k^+ are examined, which Algorithm 2 does. More precisely, it starts by visiting the simple paths starting at nodes having no predecessors (otherwise these nodes would not be visited). Once this is done, one must explore all nodes not yet marked and continue until all nodes have been visited/marked.

From the above discussion, we obtain the following theorem.

Theorem 2. *Assume one can determine in constant time for an arc (u, v) of $E_{k,c}^+$, whether u is right extensible and whether v is left extensible. Then, with the sets $Init_k$, $InitExact_k$ and $SubInit_k$, Algorithm 2 builds a graph that is isomorphic to CDBG_k^+ in linear time in the size of these sets.*

Remark. Executing Algorithm 2 does not require to build DBG_k^+, since the set of successors $\text{Succ}_k(u)$ of any node u is computed in constant time.

Characterisation of the concepts of right and left extensibility. By the construction of DBG_k^+, we get the following properties, which will turn useful for the construction of the CdBG from specific indexes (Section 3 and 4).

Proposition 6. *Let w be a word of $Init_k$. $first_k(w)$ is right extensible if and only if $|w| > k$ or $\sharp(RC(w) \cap \Sigma) = 1$.*

Proposition 7. *Let w be a word of $Init_k$ such that $first_k(w)$ is right extensible. Let the letter a be the unique element of $RC(first_k(w)) \cap \Sigma$, then $last_{k-1}(first_k(w))a$ is left extensible if and only if*

$$\sharp(Support(first_k(w))) = \sharp(Support(last_{k-1}(first_k(w))a) \setminus \{(i,1) \mid 1 \le i \le n\}).$$

Proof. Let (i, j) be a pair of $Support(first_k(w))$. We have

$$(i, j+1) \in Support(last_{k-1}(first_k(w))).$$

As $Support(last_{k-1}(first_k(w))) = Support(last_{k-1}(first_k(w))a)$, it follows that

$$(i, j+1) \in Support(last_{k-1}(first_k(w))a).$$

If there exists $(i, j) \in Support(last_{k-1}(first_k(w)))$ such that $j > 0$ and $(i, j-1) \notin Support(first_k(w))$, there exists a letter $b \neq w[1]$ such that $(i, j-1) \in Support(b \cdot last_{k-1}(first_k(w)))$.

Hence $(b \cdot last_{k-1}(first_k(w)), last_{k-1}(first_k(w))a)$ also belongs to E^+, and thus $last_{k-1}(first_k(w))a$ is not left extensible. □

In summary, this section gives a formulation of the dBG of S in terms of words. Now assume that the substrings of the words are indexed in a data structure, *e.g.* a generalised suffix array. How can we build the dBG or the contracted graph directly from this structure? To achieve this, it suffices to compute the three sets $Init_k$, $InitExact_k$, $SubInit_k$, as well as the sets $Support(.)$ and $\mathrm{Succ}_k(.)$ for some appropriate substrings. In the following sections, we exhibit algorithms to compute DBG_k^+ and CDBG_k^+ for two important indexing structures and for a home-made truncated data structure.

3. Transition from an indexing data structure to de Bruijn graphs

3.1. From a generalised suffix tree

Suffix Trees (ST) belong to the most studied indexing data structures. A *generalised* ST can index the substrings of a set of words. Generally for this sake, all words are concatenated and separated by a special symbol not occurring elsewhere. However, this trick is not compulsory, and an alternative is to keep the indication of a terminating node within each node.

3.1.1. The suffix tree and its properties

The *Generalised Suffix Tree* of a set of words S is the suffix tree of S, where each word of S does not necessarily finish by a letter of unique occurrence. Hence, for each node v of the Generalised Suffix Tree of S, we keep in memory the set, denoted by $Suff(v)$, of pairs (i, j) such that the word represented by v is the suffix of s_i starting at position j. Let us denote by T the generalised suffix tree of S (from now on, we simply say the tree) and by V_T its set of nodes. For $v \in V_T$, $Children(v)$ denotes its set of children and $f(v)$ its parent. See Fig. 5 for an example of GST.

Some nodes of T may have just one child. The size of the union of $Suff(v)$ for all node v of T equals the number of leaves in the generalised suffix tree when the words end with a terminating symbol. Hence, the space to store T and the sets $Suff(.)$ is linear in $\|S\|$. By simplicity, for a node v of T, the word represented by v is confused with v. For each node v of T, $v \in F(S)$. As all elements of $F(S)$ are not necessarily represented by a node of T, we give the following proposition.

Proposition 8. *The set of nodes of T is exactly the set of words w of $F(S)$ such that $d(w) = 0$.*

We recall the notion of a suffix link (SL) for any node v of T (leaves included). Let $sl(v)$ denote the node targeted by the suffix link of v, *i.e.*, $sl(v) = v[2..|v|]$. By definition of a suffix tree, for all $w \in F(S)$, there exists a node v of T such that w is a prefix of v. Let v' the node of minimal length of T such that w is a prefix of v, then $|v'| = |w| + d(w)$, and therefore $\lceil w \rceil = v'$.

Proposition 9. *Let $w \in F(S)$. Then $|\lceil w \rceil| \ge |w| > |f(\lceil w \rceil)|$, where $f(\lceil w \rceil)$ is the* parent *of $\lceil w \rceil$ in T.*

Proof. As $f(\lceil w \rceil) = \lfloor w \rfloor$, the result is obvious. □

3.1.2. Construction of DBG_k^+

Let $[x_1..x_m]$ be the set of k-mers of S. According to the definition of $Init_k$ and to Proposition 4, $Init_k = [\lceil x_1 \rceil..\lceil x_m \rceil]$. Thus, by Proposition 9, $Init_k = \{v \in V_T \mid |f(v)| < k \text{ and } |v| \ge k\}$. Similarly, $InitExact_k = \{v \in V_T \mid |v| = k\}$. Now, it appears clearly that $InitExact_k$ is a subset of $Init_k$, since for all $v \in V_T$, $|f(v)| < |v|$.

We consider the same two cases as for the construction of E^+ on p. 6, but in the case of a tree. Let $v \in Init_k$.

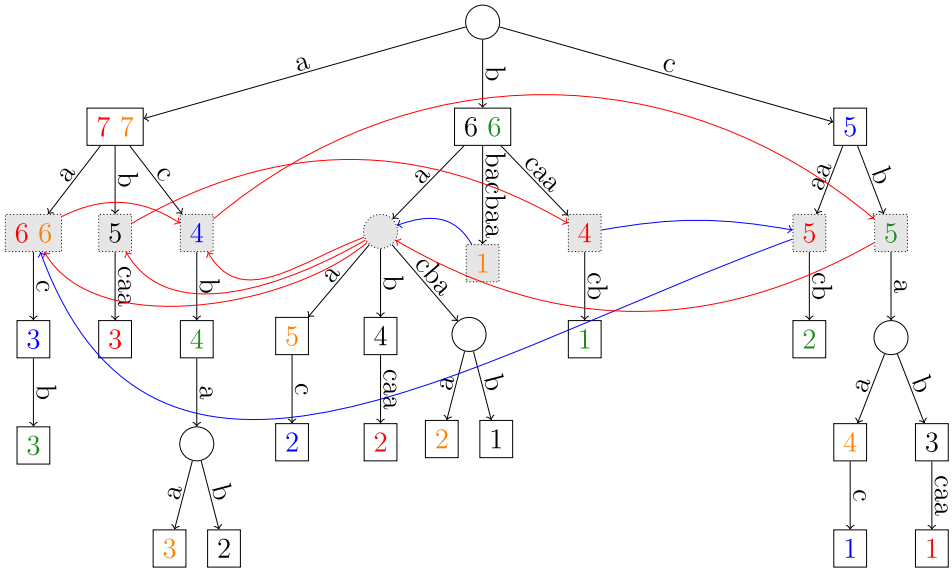

Fig. 5. The generalised suffix tree for our running example and the constructed de Bruijn graph for $k := 2$. Square nodes represent words that occur as a suffix of some s_i, circle nodes are the other nodes of T. Nodes in grey are those used to represent the nodes of the dBG. Each square node stores its positions of occurrences in S; for simplicity, we display the starting position as a number and the word of S in which it occurs as its colour, instead of showing the list of pairs (i, j). The solid curved arrows are the edges of the de Bruijn graph for $k := 2$; those coloured in red correspond to Case 1 and those in blue to Case 2. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

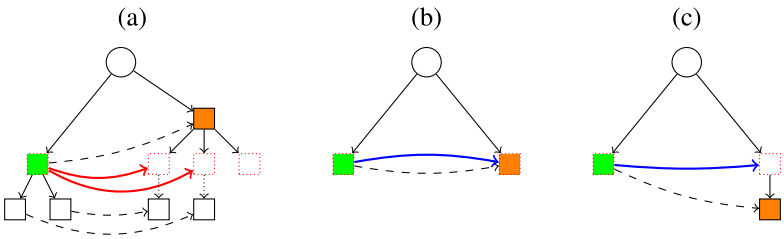

Fig. 6. The figures (a), (b) and (c) show Case 1 and Case 2 encountered when computing the arcs of DBG_k^+. The green node represents the node v, and the one in orange $sl(v)$. The dashed arcs correspond to suffix links. Arcs of DBG_k^+ are in solid line and coloured in red for Case 1 (a), or in blue for Case 2 (b), (c). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Case 1. $|v| = k$ (Fig. 6a).

As $v \in InitExact_k$, $sl(v) \in SubInit_k$. Therefore, each child u of $sl(v)$ is an element of $Init_k$. Thus, the outgoing arcs of v in DBG_k^+ are the arcs from v to the child u of $sl(v)$ where the first letter of the label between $sl(v)$ and u is an element of the right context of v. As the set of the first letters of the label between v and children of v is exactly $RC(v) \cap \Sigma$, the number of outgoing arcs of v in DBG_k^+ is the number of children of v. To build the outgoing arcs of v in DBG_k^+, for each child u' of v, we associate v with the node of $Init_k$ between the root and $sl(u')$, *i.e.*, $\lceil first_k(sl(u'))\rceil$.

Case 2. $|v| > k$ (Figs. 6b and 6c).

We have that $sl(v)$ is a node of V_T. As $|v| > k$, $|sl(v)| \geq k$. Thus, there exists an element of $Init_k$ between the root and $sl(v)$. We associate v with this node, *i.e.* $\lceil first_k(sl(v))\rceil$.

We illustrate these two cases in Fig. 5:

Case 1. Case where v is 6,6, $sl(v)$ is 7,7, the unique child u' of v is 3, and $sl(u')$ is 4, which is in $Init_k$.

Case 2. Case where v is 1, $sl(v)$ is 2, and $\lceil first_k(sl(v))\rceil$ is ◯.

In both cases, building the arcs of E^+ requires to follow the SL of some node. The node, say u, pointed at by a SL may not be initial. Hence, the initial node representing the associated first k-mer of u is the only ancestral initial node of u. We

equip each such node u with a pointer $p(u)$ that points to the only initial node on its path from the root. In other words, for any $u \notin Init_k$ such that $|u| > k$, one has $p(u) := \lceil first_k(u) \rceil$.

The algorithm to build the DBG_k^+ is as follows. An initial depth first traversal of T allows to collect the nodes of $Init_k$ and for each such node to set the pointer $p(.)$ of all its descendants in the tree. Finally to build E^+, one scans through $Init_k$ and for each node v one adds $\mathrm{Succ}_k(v)$ to E^+ using the formula given above. Altogether this algorithm takes a time linear in the size of T. Moreover, the number of arcs in E^+ is linear in the total number of children of initial nodes. This gives us the following result.

Theorem 3. *For a set of words S, building the de Bruijn Graph of order k, DBG_k^+ takes linear time and space in $|T|$,* i.e., *in $\|S\|$.*

3.1.3. Construction of $CDBG_k^+$

In Section 2.3, we have seen an algorithm that allows to compute directly CDBG_k^+ provided that one can determine if a node v is right extensible and if $\mathrm{next}(v)$ is left extensible, where $\mathrm{next}(v)$ denotes the only successor of v. Let us see how to compute the extensibility in the case of a Suffix Tree.

By applying Proposition 6 in the case of a tree, for an element v of $Init_k$, $first_k(v)$ is right extensible if and only if $|v| > k$ or $\sharp(Children(v)) = 1$. Thus checking the right extensibility of a node takes constant time.

For the left extensibility of the single successor of a node, one only needs the size of support of some nodes (Proposition 7). Let us see first how to compute $\sharp(Support(.))$ on the tree, and then how to apply Proposition 7.

Proposition 10. *Let v be a word of $F(S)$ and $V_T(\lceil v \rceil)$ denotes the set of nodes of the subtree rooted in $\lceil v \rceil$.*

$$Support(v) = \bigcup_{v' \in V_T(\lceil v \rceil)} Suff(v').$$

Along a traversal of the tree, we can compute and store $\sharp(Support(v))$ and $\sharp(Support(v) \cap \{(i,1) \mid 1 \le i \le n\})$ for each node v in linear time in $|T|$.

Let v be a word of $Init_k$ such that $first_k(v)$ is right extensible.

Case 1. If $|v| = k$, then $first_k(v) = v$ and $\sharp(Children(v)) = 1$. Let u be the only child of v. Thus, $|u| > k$, $RC(v) \cap \Sigma = \{u[k+1]\}$, and $last_{k-1}(v)u[k+1] = first_k(sl(u))$. Hence,

$$\sharp(Support(v)) = \sharp(Support(first_k(sl(u))) \setminus \{(i,1) \mid 1 \le i \le n\})$$

and by Proposition 7, $first_k(sl(u))$ is left extensible.

Case 2. If $|v| > k$, then $RC(first_k(v)) \cap \Sigma = \{v[k+1]\}$ and

$$last_{k-1}(first_k(v))v[k+1] = last_k(first_{k+1}(v)) = first_k(sl(v)).$$

By Proposition 7, $first_k(sl(v))$ is left extensible if and only if

$$\sharp(Support(first_k(v))) = \sharp(Support(first_k(sl(v))) \setminus \{(i,1) \mid 1 \le i \le n\})$$

As $\sharp(Support(first_k(v))) = \sharp(Support(\lceil first_k(v) \rceil))$ and $\sharp(Support(v) \setminus \{(i,1) \mid 1 \le i \le n\}) = \sharp(Support(v)) - \sharp(Support(v) \cap \{(i,1) \mid 1 \le i \le n\})$, determining if $\mathrm{next}(v)$ is left extensible takes constant time. To conclude, as for any initial node v, we can compute in $O(1)$ time its set of successors $\mathrm{Succ}_k(v)$, its right extensibility, and the left extensibility of its single successor, we can readily apply Algorithm 2 to built CDBG_k^+ and we obtain a complexity that is linear in the size of DBG_k^+, since each successor is accessed only once. This yields Theorem 4.

Theorem 4. *For a set of words S, building the Contracted de Bruijn Graph of order k, $CDBG_k^+$ takes linear time and space in $|T|$,* i.e., *in $\|S\|$.*

3.2. From a generalised suffix array

In the previous subsections we have shown how to build de Bruijn graphs from suffix trees. Suffix trees are very elegant data structures but they are too space-consuming in practice. In many applications they have been replaced by suffix arrays that are equivalent data structures and are more space economical. We will now show how to build de Bruijn graphs from suffix arrays.

Let SA and LCP be the generalised enhanced suffix array of S:

- $\forall 1 \le i < \|S\|$, $SA[i] = (g,h)$, $SA[i+1] = (g',h')$ then $s_g[h..|s_g|] < s_{g'}[h'..|s_{g'}|]$,
- $\forall 2 \le i \le \|S\|$, $LCP[i]$ is the length of the longest common prefix between suffixes stored in $SA[i-1]$ and in $SA[i]$, and $LCP[1] = LCP[\|S\|+1] = -1$.

Let us recall the definition of an lcp-interval.

Definition 4 *([23]).* An interval $[i, j]$, $1 \leq i < j \leq \|S\|$ is called a lcp-interval of value ℓ, also denoted by ℓ-$[i, j]$, iff:

1. $LCP[i] < \ell$,
2. $LCP[g] \geq \ell$ for $i < g \leq j$,
3. $LCP[g] = \ell$ for at least one g such that $i < g \leq j$,
4. $LCP[j+1] < \ell$.

Let us now recall the definitions of the previous and next smaller values (PSV and NSV) arrays.

Definition 5 *([23]).* For $2 \leq i \leq \|S\|$:

- $PSV[i] = \max\{j \mid 1 \leq j < i \text{ and } LCP[j] < LCP[i]\}$,
- $NSV[i] = \min\{j \mid i < j \leq \|S\| + 1 \text{ and } LCP[j] < LCP[i]\}$.

Recall that if $2 \leq i \leq \|S\|$ then $[PSV[i], NSV[i] - 1]$ is an lcp-interval of value $LCP[i]$. The direct inclusion among lcp-intervals defines a tree relationship called the lcp-interval tree (see [23, Def. 4.4.3, p. 87]). Given an lcp-interval ℓ-$[i, j]$, its parent lcp-interval ℓ'-$[i', j']$ can be easily computed in constant time using the arrays LCP, PSV and NSV. Then:

- $Init_k$ consists of:
  - the lcp-intervals ℓ-$[i, j]$ such that $\ell \geq k$ and the parent interval ℓ'-$[i', j']$ of ℓ-$[i, j]$ is such that $\ell' < k$ (the associated string is $s_{SA[i].g}[SA[i].h \,..\, SA[i].h + \ell - 1]$);
  - the positions $SA[i'] = (g, h)$ such that i' is not contained in lcp-intervals ℓ-$[i, j]$ with $\ell \geq k$ and $h \leq |s_g| - k + 1$ (the associated string is $s_g[h \,..\, |s_g]$);
- $InitExact_k$ is composed of the lcp-intervals k-$[i, j]$;
- $SubInit_k = InitExact_{k-1}$.

Actually the lcp-interval tree does not need to be explicitly build and the sets can be computed by a single scan of the SA and LCP arrays.

For an lcp-interval ℓ-$[i, j] \in Init_k$ we have $\sharp(Support(s_{SA[i].g}[SA[i].h \,..\, SA[i].h + k - 1])) = j - i + 1$.

Theorem 5. *The de Bruijn graph of order k, $CDBG_k^+$, for a set of words S can be built in a time and space that are linear in $\|S\|$ using the generalised suffix array of S.*

4. Transition from a truncated structure to de Bruijn graphs

This section is organised as follows. In Section 4.1, we define a simple condition that a set of input strings must satisfy to allow building a generalised index and sketch a modification of McCreight's algorithm [14] for doing so. In Section 4.2, we introduce the reduced truncated suffix tree and specialise the previous algorithm for constructing it efficiently. Finally, in Section 4.3 we show how to construct both the de Bruijn Graph and its contracted version in optimal time from the reduced truncated suffix tree.

4.1. Set of chains of suffix-dependant strings and tree

Here, we introduce the notion of *suffix dependence* between strings, and the notion of *chain of suffix-dependant strings* in order to define a unified index that generalises both the suffix tree [14] and the truncated suffix tree [18]. First, let us define the concept of suffix-dependant strings and of chains of suffix-dependant strings.

Definition 6.

1. A string x is said to be *suffix-dependant* of another string y if $x[2..|x|]$ is prefix of y.
2. Let w be a string and m be a positive integer smaller than $|w| - 1$. A m-tuple of m strings $(x_1, \ldots, x_m)$ is a *chain of suffix-dependant strings of* w if x_1 is a prefix of w and for each $i \in [2, m]$, x_i is a prefix of $w[i, |w|]$ such that $|x_i| \geq |x_{i-1}| - 1$.

Let $\mathcal{R} = \{C_1, \ldots, C_n\}$ be a set of tuples such that for each $i \in [1, n]$, C_i is a chain of suffix-dependant strings of the string s_i. For $i \in [1, n]$ and $j \in [1, |C_i|]$, $C_i[j]$ is the jth string of the tuple C_i. Let $\widehat{\mathcal{R}} = \{\widehat{C_1}, \ldots, \widehat{C_n}\}$ be the set of tuples such that for each $i \in [1, n]$ and $j \in [1, |C_i|]$, $\widehat{C_i}[j] = |C_i[j]|$, *i.e.* $\widehat{\mathcal{R}}$ contains tuples of lengths.

With $\widehat{\mathcal{R}}$ and S, we can easily compute $\mathcal{R}$. In the sequel, we use $\mathcal{R}$ to demonstrate our results, and $\widehat{\mathcal{R}}$ to state the complexities of algorithms. Indeed, in the case where C_i is the tuple of each suffix of s_i, the size of C_i is linear in $|s_i|^2$ but $\widehat{C_i}$ is linear in $|s_i|$.

Let w be a string; w may occur in distinct tuples of $\mathcal{R}$. Thus, we define $N(w)$ the set of (i, j) such that $w = C_i[j]$. In other words, $N(w)$ is the set of coordinates of the elements of $\mathcal{R}$ that are equal to w.

We define a contracted version of the well-known Aho–Corasick tree [17]. In fact, we apply nearly the same contraction process that turns a trie of a word into its compact Suffix Tree [17]. Consider the Aho–Corasick tree of S, in which each node represents a prefix of words in S. We contract the non-branching parts of the branches except that we keep all nodes representing a word that belongs to a tuple in $\mathcal{R}$. From now on, let $T(\mathcal{R})$ denote this contracted version of the Aho–Corasick tree of S.

$\mathcal{N}$ and $\mathcal{L}$ denote respectively the set of nodes and the set of leaves of $T(\mathcal{R})$. Furthermore, we define for each node v of $T(\mathcal{R})$ two weights:

- $s(v)$ is the number of times that an element of a tuple of $\mathcal{R}$ is equal to the word represented by v (*i.e.*, $s(v) := |N(v)|$).
- $t(v)$ is the number of times that the first element of a tuple of $\mathcal{R}$ is equal to the word represented by v (*i.e.*, $t(v) := |\{(i, 1) \in N(v) \mid i \in [1, n]\}|$).

Let w be a string, we put $Succ(w) = \{(i, j) \mid (i, j-1) \in N(w) \text{ and } j \leq |C_i|\}$. We define $\mathcal{H}$ as the subset of $\mathcal{L}$ such that:

$$\mathcal{H} := \{u \in \mathcal{L} \mid \exists C \in \mathcal{R} \text{ and } j < |C| \text{ such that } u = C[j]\}$$

It is equivalent to say that $\mathcal{H} = \{u \in \mathcal{L} \mid Succ(u) \text{ is not empty}\}$. A mapping m from $\mathcal{H}$ to $\mathcal{N}$ is called *possible link* if for each node v in $\mathcal{H}$, $\exists (i, j) \in Succ(v)$ such that $m(v) = C_i[j]$.

Below we present an algorithm that constructs $T(\mathcal{R})$, and computes for each node v in $\mathcal{N}$, the weights $s(v)$ and $t(v)$ and a possible link P_0.

Construction of $T(\mathcal{R})$. Now, we give an algorithm to construct $T(\mathcal{R})$. We use the version of McCreight's algorithm given by Na et al. [18] on our input and we build for each leaf v, $s(v)$, $t(v)$ and $P_0(v)$. For building $T(\mathcal{R})$, we start with a tree that contains only the root. Then, for each word w in every chain C, we create or update (if it exists) the node w as follows. Assume that we keep in memory the node v that has been processed just before w.

If w is the first word of C, we go down from the root by comparing w to the labels of the tree. If we create the node w, $s(w)$ and $t(w)$ are initialised to 1, and $P_0(w)$ to *nil*. If w already exists on the tree, we increment $s(w)$ and $t(w)$ by 1.

If w is not the first word of C, we start from v, and as in McCreight's algorithm, we create or arrive on the node representing w. If we need to create this node, $s(w)$ is initialised to 1, $t(w)$ to 0, and $P_0(w)$ to *nil*. Otherwise, we add 1 to $s(w)$. We set $P_0(v) = w$.

The loop continues with the next word until the end, and we obtain $T(\mathcal{R})$.

Theorem 6. *For a set of chain of suffix-dependant strings $\mathcal{R}$, we can construct $T(\mathcal{R})$ in $O(\|S\|)$ time and space.*

Proof. To begin with, let us to prove that $T(\mathcal{R})$ is in $O(\|S\|)$ space. Its number of leaves equals $\sum_{C \in \mathcal{R}} |C|$. Hence, its number of nodes is at most $2\sum_{C \in \mathcal{R}} |C| - 1 \leq 2\|S\|$, and its number of edges is at most $2\|S\|$. Thus the size of $T(\mathcal{R})$ is in $O(\|S\|)$.

Clearly, the construction algorithm of $T(\mathcal{R})$ computes both weights $s(.)$ and $t(.)$, and the possible link $P_0(.)$ correctly. For the complexity, for each chain of suffix-dependant C_i of $\mathcal{R}$, the length of the traverse path on the tree is equal to $|w_i|$, thanks to the use of the suffix links. Thus as in McCreight's algorithm, the complexity is in $O(\|S\|)$. □

Now, we are equipped with an algorithm that builds $T(\mathcal{R})$ for any set of chains of suffix-dependant strings. Let us review some instances of sets S, for which $T(\mathcal{R})$ is in fact a well-known tree.

- If $\mathcal{C} := \cup_{w \in S}\{\text{tuple of suffixes of } w\}$, then $T(\mathcal{C})$ is the Generalised Suffix Tree of S (see Fig. 7a). We have that the restrained mapping $sl(.)$ is an example of a possible link.
- If $B_k := \cup_{w \in S}\{\text{tuple of } k\text{-mer of } w \text{ and suffixes of length } k' < k \text{ of } w\}$, then $T(B_k)$ is the generalised k-truncated suffix tree of S, as defined in [19] (which generalises the k-truncated suffix tree of Na et al. [18]).
- If $A_k := \cup_{w \in S}\{\text{tuple of } k+1\text{-mer of } w \text{ and suffixes of length } k \text{ of } w\}$, then $T(A_k)$ is the truncated suffix tree that we define below in Section 4.2 (see Fig. 7b).

4.2. Our truncated suffix tree

First, we define the following notation.

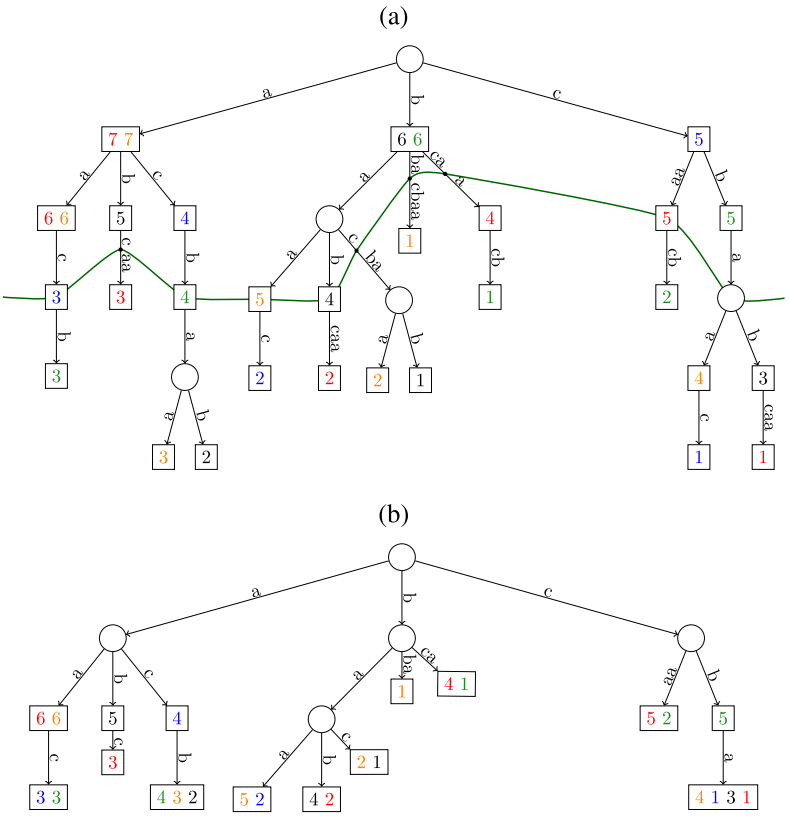

Fig. 7. (a) The generalised suffix tree for the set of words {*bacbab*, *bbacbaa*, *bcaacb*, *cbaac*, *cbabcaa*}. The part above the green line corresponds to the TST $T(A_2)$, which is shown in (b). (b) The truncated suffix tree $T(A_2)$ for the same set of words. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Definition 7.

1. For all $i \in [1, |S|]$ and $j \in [1, |s_i| - k + 1]$, $A_{k,i}$ denotes the tuple such that its jth component is defined by

$$A_{k,i}[j] := \begin{cases} w_i[j, j+k] & \text{if } j \le |w_i| - k \\ w_i[j, |w_i|] & \text{otherwise} \end{cases}$$

2. and A_k is the set of these tuples: $A_k := \bigcup_{i=1}^{n} A_{k,i}$.

Proposition 11.

1. *$A_{k,i}$ is a chain of suffix-dependant strings of s_i.*
2. *Moreover, $\{w \in A_{k,i} \mid A_{k,i} \in A_k\} = F_{k+1}(S) \cup Suff_k(S)$.*

Proof.

1. For all $j \in [1, |A_{k,i}| - k]$, it is easy to see that $A_{k,i}[j]$ is a suffix-dependant string of $A_{k,i}[j+1]$.
2. For the second point

$$\{w \in A_{k,i} \mid A_{k,i} \in A_k\} = \bigcup_{i=1}^{n} \Big(\bigcup_{j=1}^{|s_i|-k+1} \{A_{k,i}[j]\} \Big)$$

$$= \bigcup_{i=1}^{n} \Big(\bigcup_{j=1}^{|s_i|-k} \{A_{k,i}[j]\} \bigcup \{A_{k,i}[|s_i| - k + 1]\} \Big)$$

♯reads	100	1000	10000
ST	14382	135558	1320811
TST ($k=5$)	1352 (9.40%)	1365 (1.00%)	1365 (0.10%)
TST ($k=10$)	14100 (98.03%)	120602 (88.96%)	677153 (51.26%)
TST ($k=20$)	14347 (99.75%)	133204 (98.26%)	1263803 (95.68%)
TST ($k=40$)	14382 (100.00%)	134316 (99.08%)	1291685 (97.79%)

♯reads	100000	1000000	2249632
ST	12354838	103555389	216725799
TST ($k=5$)	1365 (0.01%)	1365 (0.001%)	1365 (0.0006%)
TST ($k=10$)	1315886 (10.65%)	1396675 (1.34%)	1397752 (0.64%)
TST ($k=20$)	10549607 (85.38%)	49389538 (47.69%)	69248532 (31.95%)
TST ($k=40$)	11337038 (91.76%)	69375578 (66.99%)	117282522 (54.11%)

Fig. 8. Number of nodes of the GST vs the TST for $k = 5, 10, 20, 40$ and the percentage compared to the GST for Illumina reads of length 101.

$$= \bigcup_{i=1}^{n} (F_{k+1}(\{s_i\}) \bigcup Suff_k(\{s_i\}))$$
$$= F_{k+1}(S) \cup Suff_k(S) \qquad \square$$

By applying the algorithm described in Section 4.1 to the set A_k (Definition 7), and by using Theorem 6, we get the following result.

Corollary 1. *We can construct $T(A_k)$ in $O(\|S\|)$ time and space.*

4.2.1. Experimental results

We tested the two data structures GST and TST on real biological data. We considered a set of 2249632 Illumina reads of yeast of length 101 and performed tests for subsets of size 100, 1000, 10000, 100000, 1000000 and for the whole set. We counted the number of nodes of the GST and of the TST for various values of k (5, 10, 20 and 40). We used the `gsuffix`[1] of [19]. It should be noted that their implementation of the TST stores all the suffixes shorter than k producing thus more nodes than our TST. Fig. 8 displays the results. It can be seen that for small sets, TSTs do not save many nodes compared to the GST except for very small values of k but that for large sets TSTs save a lot of nodes for small values of k, they save more than two third of nodes for $k = 20$ and almost half of the nodes for $k = 40$. We also performed experiments with longer reads from Pacific Biosciences technology (not shown here). In this case, as expected, TSTs save less nodes than for Illumina reads.

4.3. De Bruijn graph via the truncated suffix tree

Here, we describe an algorithm that builds the de Bruijn Graph of S starting from the generalised truncated suffix tree of S.

4.3.1. De Bruijn graph

Proposition 12 states that there does not exist any leaf in $T(A_k)$ representing a word strictly shorter than k.

Proposition 12. *Let v be a leaf of $T(A_k)$. Then $|v| = k$ or $|v| = k + 1$.*

Proof. For all $w_i \in S$ and $j \in [1, |w_i| - k + 1]$, $|A_{k,i}[j]| = k$ or $k + 1$. $\square$

We set $Init_k = \{v \in V_{T(A_k)} \mid |v| \geq k \text{ and } |f(v)| < k\}$. For a possible link P_0, we define the mapping P from $\mathcal{H}$ to $\mathcal{N}$. $\mathcal{H}$, $\mathcal{N}$ and $\mathcal{L}$ have the same definition as before, but applied to the $T(A_k)$. $\mathcal{H}$ can be seen in this case as the set of leaves of length $k + 1$ of $T(A_k)$. We define the mapping P as follows:

$$P : \mathcal{H} \longrightarrow \mathcal{N}$$
$$v \mapsto \begin{cases} P_0(v) & \text{if } P_0(v) \in Init_k \\ f(P_0(v)) & \text{otherwise} \end{cases}$$

[1] http://gsuffix.sourceforge.net/gsuffix-docs/main.html.

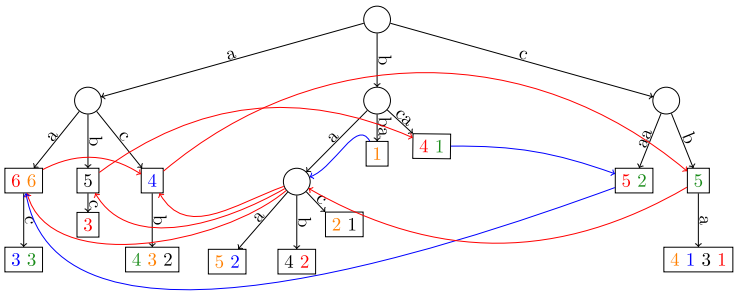

Fig. 9. The de Bruijn graph of order 2 built on $T(A_2)$. The solid curved arrows are the edges corresponding to the first part of the definition of E_k^+, while those in blue correspond to the second part. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The mapping P can be constructed in linear time in $O(\|S\|)$. In fact, for each $v \in \mathcal{H}$, $P(v)$ can be constructed in $O(1)$ time because in this case, $P_0(v) \in Init_k \Leftrightarrow |f(P_0(v))| \neq k$. As $|\mathcal{H}| \leq \|R\|$, we can construct P for all elements of $\mathcal{H}$ in $O(\|R\|)$. Indeed, it is enough to consider the length of the parent of $P_0(v)$ to decide if $P_0(v)$ is in $Init_k$.

Proposition 13. *Let $v \in \mathcal{L}$, $P(v) \in Init_k$ and $P(v) = sl(v)$ if $sl(v)$ exists.*

Proof. Let $v \in \mathcal{L}$. If $v \in \mathcal{H}$ and $P_0(v) \notin Init_k$, $|f(P_0(v))| = k$ and thus $P(v) = f(P_0(v)) \in Init_k$. According to the definitions of a possible link P, and of A_k, for any node v in $\mathcal{L}$, $P(v)$ is the shortest node of $T(A_k)$ such that v is a prefix of $P(v)$. Hence, $P(v) = sl(v)$. □

Theorem 7. *We can construct DBG_k^+ in $O(\|S\|)$ time and in $O(size(DBG_k^+))$ space.*

Proof. We begin by building $T(A_k)$. With $T(A_k)$, we can build $Init_k$, $SubInit_k$ and $InitExact_k$ as we do on the generalised suffix tree of S. By using P as the suffix link, we can build the graph (V, E) satisfying

$$V = Init_k,$$
$$E = \left(\bigcup_{v \in Init_k, |v|=k+1} (v, P(v)) \right) \bigcup \left(\bigcup_{v \in Init_k, |v|=k} \left(\bigcup_{u \in Children(v)} (v, P(u)) \right) \right).$$

Let us note that two cases of arcs arise depending on whether the starting node v represents a word of length k or of length $k+1$. These cases correspond to the two terms in the union above.

This graph is isomorphic to DBG_k^+.

Let b be the application from $\mathcal{H}$ to E such that

$$b(v) = \begin{cases} (v, P(v)) & \text{if } |f(v)| \neq k \\ (f(v), P(v)) & \text{if } |f(v)| = k. \end{cases}$$

As (V, E) is isomorphic to DBG_k^+, b is a bijection. As $|\mathcal{L} \setminus \mathcal{H}| \leq |S|$, $|\mathcal{L}|$ is linear in the size of DBG_k^+. □

Fig. 9 shows an example of de Bruijn graph of order 2 built from $T(A_2)$.

4.3.2. A contracted de Bruijn graph

Proposition 14. *For each leaf v of $T(A_k)$, $s(v)$ is the size of the support of v in S and $t(v)$ is the size of the set $\big(Support(v) \cap \{(i, 1) \mid 1 \leq i \leq n\}\big)$.*

Hence, we obtain the following theorem.

Theorem 8. *We can construct $CDBG_k^+$ in $O(\|S\|)$ time and in $O(size(DBG_k^+))$ space.*

Instead of using $T(A_k)$ to build DBG_k^+ or $CDBG_k^+$, we could have taken $T(B_{k+1})$. Indeed, $T(B_{k+1})$ is the tree $T(A_k)$ with additional leaves representing all suffixes shorter than $(k-1)$ of the words in S. These leaves make $T(B_{k+1})$ linear in $\|S\|$, but not in the size of DBG_k^+.

5. Dynamically updating the order of $DBG^+_{\cdot}$

Genome assembly from short reads is difficult and in practice requires to test multiple values of k for the dBG. Indeed, the presence of genomic repeats makes some orders k appropriate to assemble non-repetitive regions, and larger orders necessary to disentangle (at least some) repeated regions. Combining the assemblies obtained from DBG^+_k for successive values of k is the key of IDBA assembler, but the dBG is rebuilt for each value [8]. Other tools also exploit this idea [24]. It is thus interesting to dynamically change the order of the dBG.

An application of the BOSS structure introduced by Bowe et al. [6] to store efficiently the DBG^+_k was proposed to update dynamically the order from k to $k-1$ [25]. Their algorithm solves only the case of a decreasing k and can be used iteratively to build the dBG for all values of k in the desired range $[k_{max}, k_{min}]$. However, when the graph is built for k_{max} and iteratively updated until k_{min} as proposed, then the k-mers corresponding to reads of length between k_{max} and k_{min} will not be included in the dBG. In contrast, we propose a dynamic update for a decreasing and an increasing k, and our solution cannot forget some k-mers.

Here, we argue that starting the construction from an index instead of the raw sequences eases the update. On p. 8, we stated which information is needed in general to build DBG^+_k. Assume the words are indexed in a suffix tree T (as in Section 3.1.2).

5.1. Updating the sets of nodes

Our construction algorithm for DBG^+_k is based on three subsets of nodes of the suffix tree: $Init_k$, $InitExact_k$ and $SubInit_k$. We first explain how these three sets can be updated when the order of the dBG changes.

Consider first changing k to $k-1$. First, only the nodes of $Init_k$ whose parent represents a word of length $k-1$ are substituted by their parent in DBG^+_{k-1}; all other nodes remain unchanged. Thus, any arc of order k either stays as such or has some of its endpoints shifted toward the parent node in T. In any case, updating an arc depends only on the nature of its nodes in DBG^+_{k-1} (whether they belong to $Init_{k-1}$ or $InitExact_{k-1}$), and can be computed in constant time.

When the order decreases from k to $k-1$, we have:

- $InitExact_{k-1} = SubInit_k$ by definition.
- $Init_{k-1} \setminus InitExact_{k-1} = \{v \in Init_k \mid |f(v)| < k-1\}$.
  Indeed, a node of $Init_{k-1}$ either belongs to $InitExact_{k-1}$ (meaning its length equals $k-1$) or it already belongs to $Init_k$ and its parent is strictly shorter than $k-1$.
- $SubInit_{k-1} = \{f(v) \mid v \in Init_{k-1} \text{ and } |f(v)| = k-2\} \cup Suff_{k-2}(S)$. Obviously, a node of length $k-2$ is either a parent of a node in $Init_{k-1}$ or a leaf of length $k-2$.

The same situation arises when changing k to $k+1$. First, only nodes of $InitExact_k$ change in DBG^+_{k+1}: they are substituted by their children. Updating an arc also depends on the nature of its nodes: it can create a fork towards the children of the destination node if the latter changes, or it can be multiplied and join each child of the source to one child of the destination if both nodes change. Then, the label of the children in T indicates which children to connect to. It can be seen that updating from DBG^+_k to DBG^+_{k+1} in either direction takes linear time in the size of T. Moreover, as updating the support of nodes in T is straightforward, we can readily apply the contraction algorithm to obtain CDBG^+_{k+1} (see Section 3.1.3).

When the order increases from k to $k+1$, we have:

- $SubInit_{k+1} = InitExact_k$ by definition.
- $Init_{k+1} = (Init_k \setminus InitExact_k) \cup \bigcup_{v \in InitExact_k} Children(v)$.
  Indeed, any node of $Init_k$ that is longer than k remains in $Init_{k+1}$. Moreover, we add the children of the nodes in $InitExact_k$.
- $InitExact_{k+1} = \{v \in Init_{k+1} \mid |v| = k+1\}$.

5.2. An algorithm for a dynamic update

Here, we present the algorithms for updating the order of the dBG. We consider both cases: decreasing or increasing the order of the dBG. We detail the case where the update changes the order from k to $k-1$ using $T(A_k)$ (Algorithm 3); the opposite case, when k is increased by one, follows a similar logical scheme as explained in Section 5.1 (Algorithm 4). For the latter, we assume that the suffix tree T (or at least the reduced truncated suffix tree $T(A_{k+1})$) is in memory. However, both Algorithms 3 and 4 show that one does not need to scan again through the complete set of reads.

The input is DBG^+_k and we wish to compute DBG^+_{k-1}. Precisely, the input consists in the three subsets of nodes, $SubInit_k$, $InitExact_k$, and $Init_k$ (whose update has been detailed above), and in the application $\text{Succ}_k(.)$, which for any node of DBG^+_k gives its successors. Actually, the subsets $SubInit_k$ and $InitExact_k$ are included to allow iterating the algorithm. For simplicity and without loss of generality, we assume that k is smaller than the length of the shortest input word of S. Moreover, the algorithm stores several sets and uses an insertion procedure denoted Set.ins(*node*), which inserts a node in the set if it is

Algorithm 3: Dynamic update of the dBG from order k to $k-1$.

```
   Input  : The sets SubInit_k, InitExact_k and Init_k and the application Succ_k(.)
   Output: The sets SubInit_{k-1}, InitExact_{k-1} and Init_{k-1} and the application Succ_{k-1}(.)
 1 begin
 2   Init_{k-1} := ∅; SubInit_{k-1} := ∅; InitExact_{k-1} := SubInit_k;
 3   for v ∈ InitExact_{k-1} do  Succ_k(v) := ∅;
 4   for v ∈ Init_k do
 5     if |f(v)| = k − 1          // f(v) is already in InitExact_{k-1} then
 6       s := f(v)
 7       SubInit_{k-1}.ins(sl(s))
 8     else
 9       s := v
10       Succ_{k-1}(s) := ∅
11     Init_{k-1}.ins(s)
12     if |f(s)| = k − 2 then  SubInit_{k-1}.ins(f(s));
13     for w ∈ Succ_k(s) do
14       if |f(w)| = k − 1 then
15         t := f(w)
16         SubInit_{k-1}.ins(sl(t))
17       else
18         t := w
19       Init_{k-1}.ins(t)
20       Succ_{k-1}(s).ins(t)          // (first_{k-1}(s), first_{k-1}(t)) ∈ E^+_{k-1}
21       if |f(t)| = k − 2 then  SubInit_{k-1}.ins(f(t));
22   return SubInit_{k-1}, InitExact_{k-1}, Init_{k-1} and Succ_{k-1}(.)
```

Algorithm 4: Dynamic update of the dBG from order k to $k+1$.

```
   Input  : The sets SubInit_k, InitExact_k and Init_k and the application Succ_k(.)
   Output: The sets SubInit_{k+1}, InitExact_{k+1} and Init_{k+1} and the application Succ_{k+1}(.)
 1 begin
 2   Init_{k+1} := ∅; SubInit_{k+1} := InitExact_k; InitExact_{k+1} := ∅;
 3   for v ∈ Init_k do
 4     if |v| = k // v is in InitExact_k then
 5       S := Children(v)
 6     else
 7       S := {v}
 8     for s ∈ S do
 9       Init_{k+1}.ins(s)
10       Succ_{k+1}(s) := ∅
11       if |s| = k + 1 then  InitExact_{k+1}.ins(s) ;
12       for w ∈ Succ_k(s) do
13         if |w| = k then
14           R := Children(w)
15         else
16           R := {w}
17         for r ∈ R do
18           Succ_{k+1}(s).ins(r)
19   return SubInit_{k+1}, InitExact_{k+1}, Init_{k+1} and Succ_{k+1}(.)
```

not already in it. One can store the membership of all nodes of each set with binary vectors of length E_k^+, which is linear in the size of E_k^+.

Algorithm 3 scans through the nodes of $Init_k$ (for loop of line 4), builds the set $Init_{k-1}$ and computes the arcs of E_{k-1}^+ according to the typology of the nodes, and as a by-product it obtains $SubInit_{k-1}$ and $InitExact_{k-1}$. For each node, it determines which node is the source (denoted by s) of an arc of E_{k-1}^+ (lines 5–10), then it inserts the source node in $Init_{k-1}$, and updates $SubInit_{k-1}$ (lines 11–12). Then it loops over all successors of the source in E_k^+ (line 13), and determines which node is the target of an arc going out of s in E_{k-1}^+ (lines 14–18 – the target is denoted by t). Finally, it inserts the target in $Init_{k-1}$, the arc from s to t in $\mathrm{Succ}_{k-1}(s)$, and updates $SubInit_{k-1}$ (lines 19–21).

The correctness of the computation of $Init_{k-1}$, $InitExact_{k-1}$, and of $\mathrm{Succ}_{k-1}(.)$ follows from the section above and from the correctness of the construction of DBG_k^+ (see Section 3.1.2). Let us explain why $SubInit_{k-1}$ is correctly computed. Let

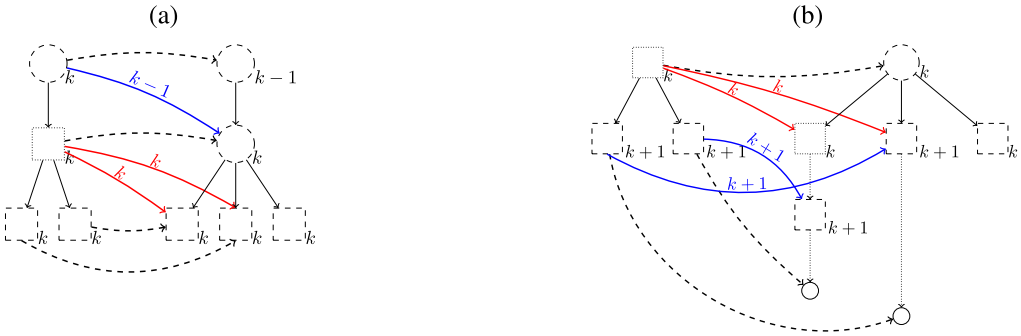

Fig. 10. Illustrating an update of k, the dBG order: (a) from k to $k-1$ and (b) from k to $k+1$. This figure shows only the cases of a type 1 node that is not right extensible. At order k, the square node in dotted line has two possible right extensions: two (red) arcs leaving it. In (a), at order $k-1$ its parent belongs to $Init_{k-1}$ and it becomes right extensible. One sees that the tree structure helps in determining this. In (b), the pendant situation occurs, where the children of the square node in dotted line belong to $Init_{k+1}$ and they both become right extensible. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

z be a node of $SubInit_{k-1}$. Assume z is a parent of some node of $Init_{k-1}$. If the latter is a source node, then z is inserted on line 12; if the latter is a target node, z is inserted on line 21. Otherwise, z must be pointed to by a suffix link of some node v. As the suffix link removes the first letter, the difference of word length between v and z is only one. Hence, v must be a node representing exactly a $(k-1)$-mer and must belong to $Init_{k-1}$. This case is detected on line 5 for a source node (line 14 for a target), and z is properly inserted on line 7 (resp. on line 16). This ends the correctness proof of Algorithm 3.

The updates of nodes in cases 1 (see Fig. 6 on p.9) are illustrated in Fig. 10. Looking at the tree rooted in $sl(s)$ and whose leaves are the $\mathrm{Succ}_k(s)$, one can determine if one faces the case illustrated in Fig. 10a when changing k to $k-1$.

Clearly, the two nested loops of Algorithm 3 scans over E_k^+. The instructions inside can all be performed in constant time. The complexity of Algorithm 3 is thus linear in the number of arcs of E_k^+. Moreover, since it outputs what it needs as input, one can iterate this algorithm over any interval of values of k. Finally, the construction algorithm that starts directly from the suffix tree is asymptotically optimal and takes the same time complexity as the dynamic update of the dBG order.

6. Conclusion and perspectives

De Bruijn Graphs (dBG) are intricate structures and intensively exploited for assembling large genomes from short sequences. Understanding their complexity can help improving their representations or traversal algorithms. We investigate algorithms to transform indexing data structures of the input words into a dBG of those words and propose linear time algorithms when starting from Suffix Trees and Suffix Arrays to build directly a contracted dBG. Although the algorithms need a slight adaptation, all results obtained are clearly valid for both definitions of the dBG: DBG_k^+ and DBG_k^-. Moreover, we show that this approach provides a way to update dynamically the graph when one changes its order k. Algorithms enabling a dynamic update represent a theoretical challenge as well as an exciting avenue for improving genome assembly methods [24,8]. Other topics for future research include transforming compressed indexes, such as a FM-index [23], into a dBG, implementing a practical contracted dBG representation for DNA taking into account k-mers and their reverse complements based on these algorithms.

Acknowledgments

This work is supported by ANR Colib'read (ANR-12-BS02-0008) and by ANR IBC (ANR-11-BINF-0002).

References

[1] N.G. de Bruijn, On bases for the set of integers, Publ. Math. (Debr.) 1 (1950) 232–242.
[2] P. Pevzner, H. Tang, M. Waterman, An Eulerian path approach to DNA fragment assembly, Proc. Natl. Acad. Sci. USA 98 (17) (2001) 9748–9753.
[3] G. Rizk, A. Gouin, R. Chikhi, C. Lemaitre, MindTheGap: integrated detection and assembly of short and long insertions, Bioinformatics 30 (24) (2014) 3451–3457.
[4] L. Salmela, E. Rivals, LoRDEC: accurate and efficient long read error correction, Bioinformatics 30 (24) (2014) 3506–3514.
[5] T.C. Conway, A.J. Bromage, Succinct data structures for assembling large genomes, Bioinformatics 27 (4) (2011) 479–486.
[6] A. Bowe, T. Onodera, K. Sadakane, T. Shibuya, Succinct de Bruijn graphs, in: WABI, in: Lect. Notes Comput. Sci., vol. 7534, 2012, pp. 225–235.
[7] R. Chikhi, G. Rizk, Space-efficient and exact de Bruijn graph representation based on a Bloom filter, Algorithms Mol. Biol. 8 (2013) 22.
[8] Y. Peng, H. Leung, S. Yiu, F. Chin, IDBA – a practical iterative de Bruijn graph de novo assembler, in: B. Berger (Ed.), Research in Computational Molecular Biology, in: Lect. Notes Comput. Sci., vol. 6044, Springer, Berlin, Heidelberg, 2010, pp. 426–440.
[9] R. Chikhi, G. Rizk, Space-efficient and exact de Bruijn graph representation based on a Bloom filter, in: WABI, in: Lect. Notes Comput. Sci., vol. 7534, Springer, 2012, pp. 236–248.
[10] L. Salmela, Correction of sequencing errors in a mixed set of reads, Bioinformatics 26 (10) (2010) 1284–1290.
[11] E.A. Rødland, Compact representation of k-mer de Bruijn graphs for genome read assembly, BMC Bioinform. 14 (2013) 313.

[12] T. Onodera, K. Sadakane, T. Shibuya, Detecting superbubbles in assembly graphs, in: A. Darling, J. Stoye (Eds.), Algorithms in Bioinformatics, in: Lect. Notes Comput. Sci., vol. 8126, Springer, 2013, pp. 338–348.
[13] J.T. Simpson, R. Durbin, Efficient construction of an assembly string graph using the FM-index, Bioinformatics 26 (12) (2010) i367–i373.
[14] E. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.
[15] A. Apostolico, The myriad virtues of suffix trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO Advanced Science Institutes, Series F, vol. 12, Springer, 1985, pp. 85–96.
[16] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[17] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[18] J.C. Na, A. Apostolico, C.S. Iliopoulos, K. Park, Truncated suffix trees and their application to data compression, Theor. Comput. Sci. 304 (1–3) (2003) 87–101.
[19] M.H. Schulz, S. Bauer, P.N. Robinson, The generalised k-truncated suffix tree for time-and space-efficient searches in multiple DNA or protein sequences, Int. J. Bioinform. Res. Appl. 4 (1) (2008) 81–95.
[20] B. Cazaux, T. Lecroq, E. Rivals, From indexing data structures to de Bruijn graphs, in: A. Kulikov, S. Kuznetsov, P. Pevzner (Eds.), Proc. of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), in: Lect. Notes Comput. Sci., vol. 8486, Springer International Publishing, Switzerland, 2014, pp. 89–99.
[21] B. Cazaux, T. Lecroq, E. Rivals, Construction of a de Bruijn graph for assembly from a truncated suffix tree, in: A. Dediu (Ed.), Proc. of the 9th Int. Conf. on Language and Automata Theory and Applications (LATA), in: Lect. Notes Comput. Sci., vol. 8977, Springer International Publishing, Switzerland, 2015, pp. 109–120.
[22] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[23] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013, 604 pp.
[24] A. Bankevich, S. Nurk, D. Antipov, A.A. Gurevich, M. Dvorkin, A.S. Kulikov, V.M. Lesin, S.I. Nikolenko, S. Pham, A.D. Prjibelski, A.V. Pyshkin, A.V. Sirotkin, N. Vyahhi, G. Tesler, M.A. Alekseyev, P.A. Pevzner, SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing, J. Comput. Biol. 19 (5) (2012) 455–477.
[25] C. Boucher, A. Bowe, T. Gagie, S.J. Puglisi, K. Sadakane, Variable-order de Bruijn graphs, in: 2015 Data Compression Conference, DCC, IEEE Computer Society Press, 2015, pp. 383–392.

Please cite this article in press as: B. Cazaux et al., Linking indexing data structures to de Bruijn graphs: Construction and update, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.06.008