From Indexing Data Structures to de Bruijn Graphs

Bastien Cazaux, Thierry Lecroq, Eric Rivals

From Indexing Data Structures to de Bruijn Graphs*

Bastien Cazaux,† Thierry Lecroq,‡ Eric Rivals†
† L.I.R.M.M. & Institut Biologie Computationnelle
Université de Montpellier II, CNRS U.M.R. 5506
Montpellier, France
‡ LITIS EA 4108 & UFR des Sciences et des Techniques,
Université de Rouen, France
cazaux@lirmm.fr, thierry.lecroq@univ-rouen.fr, rivals@lirmm.fr

20th January 2014

Abstract

New technologies have tremendously increased sequencing throughput compared to traditional techniques, thereby complicating DNA assembly. Hence, assembly programs resort to de Bruijn graphs (dBG) of k-mers of short reads to compute a set of long contigs, each being a putative segment of the sequenced molecule. Other types of DNA sequence analysis, as well as preprocessing of the reads for assembly, use classical data structures to index all substrings of the reads. It is thus interesting to exhibit algorithms that directly build a de Bruijn graph of order k from a pre-existing index, and especially a contracted version of the de Bruijn graph, where non branching paths are condensed into single nodes. Here, we formalise the relationship between suffix trees/arrays and dBGs, and exhibit linear time algorithms for constructing the full or contracted de Bruijn graphs. Finally, we provide hints explaining why this bridge between indexes and dBGs enables to dynamically update the order k of the graph.

1 Introduction

The de Bruijn graph (dBG) of order k on an alphabet Σ with σ symbols has σ^k vertices corresponding to all the possible distinct strings of length k on the alphabet Σ and there is a directed edge from vertex u to vertex v if the suffix of u of length $k-1$ equals the prefix of v of length $k-1$. De Bruijn graphs have various properties and are more commonly defined on all the k-mers of the strings of a finite set rather than on all the possible strings of length k on the alphabet. When a vertex u has only one outgoing edge to vertex v and when v has only one ingoing edge from vertex u then the two vertices can be merged. By applying this rule whenever possible, one gets a contracted dBG. dBGs occur in different contexts. In bioinformatics they are largely used in *de novo* assembly due to a result of Pevzner *et al* [14]. Indeed recent sequencing technologies allow to obtain hundreds of million of short sequencing reads (about 100 nucleotides long) from one DNA sample. Next step is to reconstruct the genome sequence using assembly algorithms. However, the volume of read data to process has forced the shift from the classical overlap graph approach, which requires too much memory, towards a de Bruijn Graph where vertices are k-mers of the reads. In this context, there exist compact exact data structures for storing dBGs [6, 3, 15, 4] and probabilistic data structures such as Bloom filters [12, 5]. Onodera and colleagues propose to add to the succinct dBG representation of [3] a

*This work is supported by ANR Colib'read (ANR-12-BS02-0008) and Defi MASTODONS SePhHaDe from CNRS.
†
‡

bit vector marking the branching nodes, thereby enabling them to simulate efficiently a contracted dBG, where each simple path is reduced to one edge [11].

Suffix trees are well-known indexing data structures that enable us to store and retrieve all the factors of a given string. They can be adapted to a finite set of strings and are then called generalised suffix trees. They can be built in linear time and space. They have been widely studied and used in a large number of applications (see [1] and [8]). In practice, they consume too much space and are often replaced by the more economical suffix arrays [9], which have the same properties.

Read analysis and assembly include preliminary steps like filtering and error correction. To speed up such steps, some algorithms index the substrings, or the k-mers of the reads. Hence, before the assembly starts, the read set has already been indexed and mined. For instance, the error correction software `hybrid-shrec` builds a generalised suffix tree of all reads [16]. 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 [15, 11], formal methods for constructing dBG from suffix trees are an open question. Notably, the String Graph, which is also used for genome assembly, can be constructed from a FM-index [17].

In this article, given a finite collection S of strings and an integer k we formalise the relationship between generalised suffix trees and dBGs and show how to linearly build the dBG of order k for S. Next we show how to directly build the contracted dBG of order k for S in linear time and space, without building the dBG. We also show how to perform the same task using suffix arrays. Finally, we give some hints on how to dynamically adapt our dBG construction from order k to $k-1$ or from k to $k+1$.

2 Preliminaries

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 xy the *concatenation* of x and y. For every $1 \leq i \leq j \leq |x|$, $x[i]$ denotes the i-th letter of x, and $x[i..j]$ denotes the *substring* or *factor* $x[i]x[i+1]\dots x[j]$. Let k be a positive integer. If $|x| \geq k$, $first_k(x)$ 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 $1 \leq i \leq |x|-k+1$, $(x)_{k,i}$ 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 $S = \{s_1, \dots, s_n\}$ be a finite set of words. 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.* $F_S = \{w \in \Sigma^\star \mid \exists u, v \in \Sigma^\star, 1 \leq i \leq n, s_i = uwv\}$. For a word w of F_S,

- $Support_S(w)$ is the set of pairs (i, j), where w is the substring $(s_i)_{|w|,j}$. $Support_S(w)$ is called the support of w in S.
- $RC_S(w)$ (resp. $LC_S(w)$) 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. $w'w \in F_S$).
- $\lceil w \rceil_S$ is the word ww' where w' is the longest word of $RC_S(w)$ such that $Support_S(w) = Support_S(ww')$. In other words, such that w and ww' have exactly the same support in S.
- $\lfloor w \rfloor_S$ is the word w' where w' is the longest prefix of w such that $Support_S(w') \neq Support_S(w)$.
- $d_S(w) := |\lceil w \rceil_S| - |w|$.

In other words, $\lceil w \rceil_S$ is the longest extension of w having the same support than w in S, while $\lfloor w \rfloor_S$ 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 Figure 1.

	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

Figure 1: $S := \{bacbab, cbabcaa, bcaacb, cbaac, bbacbaa\}$ is a set of words. Therefore, we have $Supports_S(ba) = \{(1,1),(1,4),(2,2),(4,2),(5,2),(5,5)\}$, $RC_S(ba) = \{\varepsilon, c, cb, cba, cbab, b, bc, bca, bcaa, a, ac, cbaa\}$, $LC_S(ba) = \{\varepsilon, c, ac, bac, b, bbac\}$ and $d_S(ba) = 0$. One has $RC_S(ba) \cap \Sigma = \{a, b, c\}$. Thus, the word ba is not right extensible in S (see Def. 2.2).

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 [7].

Definition 2.1. *Let k be a positive integer and $S := \{s_1, \ldots, s_n\}$ be a set of n words. The* de Bruijn graph *of order k for S, denoted by DBG_k^+, is a directed graph, $DBG_k^+ := (V^+, E^+)$, 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.*:*

$$V^+ := F_S \cap \Sigma^k$$

$$E^+ := \{(u,v) \in V^{+2} \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } v[k] \in RC_S(u)\}. \quad (1)$$

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

$$E^+ := \{(u,v) \in V^{+2} \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } u[1] \in LC_S(v)\}. \quad (2)$$

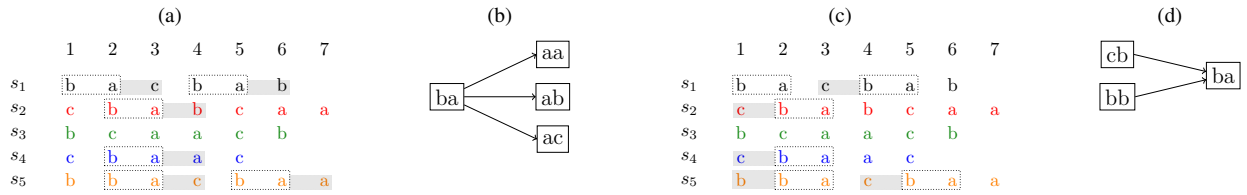

Figure 2: Examples of arcs from DBG_k^+. (a) shows letters in the right context of ba, and (b) the successors of node ba in DBG_2^+; one for each letter in $RC_S(w) \cap \Sigma$. (c) shows letters in the left context of ba, and (d) the predecessors of node ba in DBG_2^+.

Examples of arcs are displayed on Figure 2. Note that another, simpler definition of the arcs in the de Bruijn graph coexists with that of Definition 2.1. There, an arc links u to v if and only if u overlaps v by $k-1$ symbols. This graph is denoted by $DBG_k^- = (V^-, E^-)$, where:

$$V^- := F_S \cap \Sigma^k$$
$$E^- := \{(u,v) \in V^{-2} \mid last_{k-1}(u) = first_{k-1}(v)\}.$$

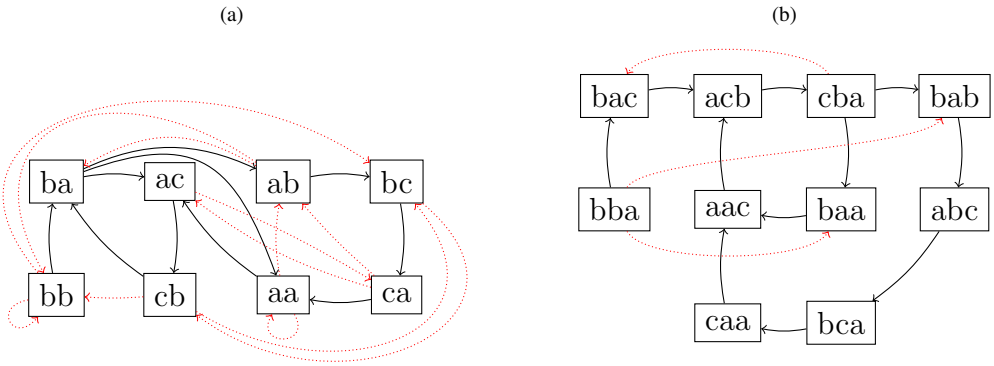

Figure 3: With solid arcs only, the graphs correspond to DBG_2^+ and DBG_3^+ for our running example. With both solid and dotted arcs, they represent DBG_2^- and DBG_3^-.

Both definitions are illustrated on Figure 3.

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

Definition 2.2 (Extensibility)**.** *Let w be a word of F_S.*

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

As S is clear from the context, we simply omit the "in S". 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.

Definition 2.3. *A* contracted de Bruijn graph *of order k, denoted by $CDBG_k^+ = (V_c^+, E_c^+)$, is a directed graph where:*

$$\begin{aligned} V_c^+ &= \{w \in \Sigma^\star \mid w \text{ is a } k'\text{-mer unique maximal by substring and } k' \geq k\} \\ E_c^+ &= \{(u,v) \in {V_c^+}^2 \mid last_{k-1}(u) = first_{k-1}(v) \text{ and } v[k] \in RC_S(last_k(u))\}. \end{aligned}$$

Note that in the previous definition, an element w in V_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 $RC_S(last_k(w)) \cap \Sigma = \{a\}$ and $last_{k-1}(w) \cdot a$ is not left extensible,
- $first_k(w)$ is not left extensible
  or $LC_S(first_k(w)) \cap \Sigma = \{a\}$ and $a \cdot first_{k-1}(w)$ is not right extensible.

With this argument, we have both following propositions.

Proposition 1. *Let $(u,v) \in E_c^+$; $(last_k(u), first_k(v)) \in E^+$ and there exists $w \in V^+$ such that $(w, first_k(v)) \in E^+ \setminus \{(last_k(u), first_k(v))\}$ or $(last_k(u), w) \in E^+ \setminus \{(last_k(u), first_k(v))\}$.*

Proposition 2. *Let $(u,v) \in E^+$. If u is right extensible and v is left extensible, then there exists $w \in V_c^+$ such that $uv[k]$ is a substring of w. Otherwise, there exists $(u',v') \in E_c^+$ 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.

3 Definition of de Bruijn Graphs with words

Let k be a positive integer. We define the following three subsets of F_S.

- $InitExact_{S,k} = \{w \in F_S \mid |w| = k \text{ and } d_S(w) = 0\}$
- $Init_{S,k} = \{w \in F_S \mid |w| \geq k \text{ and } d_S(first_k(w)) = |w| - k\}$
- $SubInit_{S,k} = InitExact_{S,k-1}$

A word of $InitExact_{S,k}$ 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 $Init_{S,k} \setminus InitExact_{S,k}$ has only one right extension.

Proposition 3. $InitExact_{S,k} = Init_{S,k} \cap \{w \in F_S \mid |w| = k\}$.

Proof Let $w \in InitExact_{S,k}$. In this case, we get $first_k(w) = w$ and $|w| - k = 0$. This means that $d_S(first_k(w)) = |w| - k$ and therefore $w \in Init_{S,k}$. □

For w an element of $Init_{S,k}$, $first_k(w)$ is a k-mer of S. Given two words w_1 et w_2 of $Init_{S,k}$, $first_k(w_1)$ 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 $Init_{S,k}$ such that $first_k(w) = w'$. From this, we get the following proposition.

Proposition 4. *There exists a bijection between $Init_{S,k}$ and the set of the k-mers of S.*

According to Definition 2.1 and Proposition 4, each vertex of DBG_k^+ can be assimilated to a unique element of $Init_{S,k}$. As the vertices of DBG_k^- are identical to those of DBG_k^+, there exists also a bijection between $Init_{S,k}$ and the set of vertices of DBG_k^-. To define the arcs between the words of $Init_{S,k}$, 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 $w \in InitExact_{S,k}$ and $a \in \Sigma \cap RC_S(w)$, there exists a unique $w' \in Init_{S,k}$ such that $last_{k-1}(w)a$ is a prefix of w'.*

Proof Let w be a word of $InitExact_{S,k}$ and a a letter of $RC_S(w)$. 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 $|last_{k-1}(w)a| + d_S(last_{k-1}(w)a) = |w'|$. By definition of $Init_{S,k}$, $w' \in Init_{S,k}$. □

The set $Init_{S,k}$ represents the nodes of DBG_k^+. Let us now build the set of arcs that is isomorphic to E^+. Let w be a word of $Init_{S,k}$ and Succ(w) denote the set of successors of $first_k(w)$: Succ$(w) := \{x \in Init_{S,k} \mid (first_k(w), first_k(x)) \in E^+\}$. We know that for each letter a in $RC_S(w)$, 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, $w \in InitExact_{S,k}$ and hence $last_{k-1}(w) \in SubInit_{S,k}$. Therefore, the outgoing arcs of w in DBG_k^+ are the arcs from w to w' satisfying the condition of Proposition 5. Then,

$$\text{Succ}(w) = \bigcup_{a \in \Sigma \cap RC_S(w)} \lceil last_{k-1}(w)a \rceil_S.$$

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 $\lceil w[2..k] \rceil_S$. Indeed, by definition of $Init_{S,k}$, $\lceil w[2..k] \rceil_S \in Init_{S,k}$, and thus

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

Now, we can build integrally DBG_k^+ or more exactly a isomorphic graph of DBG_k^+. Thus for simplicity, from now on we confound the graph we build with DBG_k^+. To do the same with CDBG_k^+, we need to characterise the concepts of right and left extensibility in terms of word properties. By the construction of DBG_k^+, we have the following results.

Proposition 6. *Let w be a word of $Init_{S,k}$. $first_k(w)$ is right extensible if and only if $|w| > k$ or $\sharp(RC_S(w) \cap \Sigma) = 1$.*

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

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

Proof Let (i,j) be a pair of $Support_S(first_k(w))$. We have $(i, j+1) \in Support_S(last_{k-1}(first_k(w)))$.
As $Support_S(last_{k-1}(first_k(w))) = Support_S(last_{k-1}(first_k(w))a)$, it follows that

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

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

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. For the sake of brevity, the algorithm and its recursive procedures are given in Appendix (see p. 8).

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 $Init_{S,k}$, $InitExact_{S,k}$, $SubInit_{S,k}$, as well as the sets $Support_S(.)$ and $\mathrm{Succ}(.)$ for some appropriate substrings. In the following sections, we exhibit algorithms to compute DBG_k^+ and CDBG_k^+ for two important indexing structures.

4 Transition from the suffix tree to de Bruijn graphs

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.

4.1 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 finish necessarily by a letter of unique occurrence. Hence, for each node v of Generalised Suffix Tree of S, we keep in memory the set, denoted by $\mathrm{Suff}_S(v)$, of pairs (i,j) such that the word represented by v is the suffix of s_i starting at position j. Let us

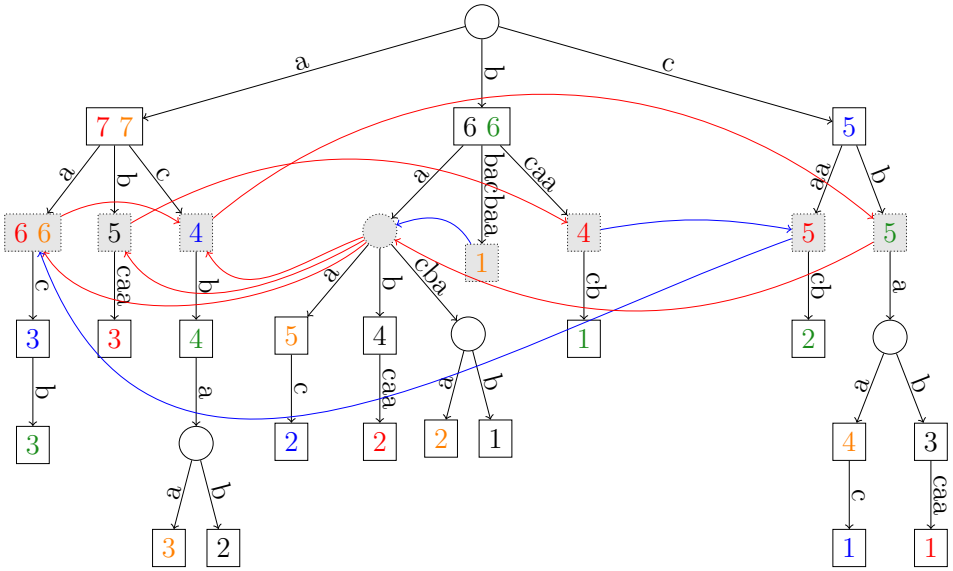

Figure 4: The Generalised Suffix Tree for our running example and the constructed De Bruijn Graph for $k := 2$. Square nodes represent words that occur as a suffix of some s_i, circle nodes are the other nodes of T. Nodes in grey are those used to represent the nodes of the dBG. Each square node stores 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 colored in red correspond to Case 1 and those in blue to Case 2 (see p. 8).

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.

Some nodes of T may have just one child. The size of the union of $\text{Suff}_S(v)$ 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 $\text{Suff}_S(.)$ 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 $d_S(w) = 0$.*

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 $|v'| = |w| + d_S(w)$, and therefore $\lceil w \rceil_S = v'$.

Proposition 9. *Let $w \in F_S$. Then $|\lceil w \rceil_S| \geq |w| > |f(\lceil w \rceil_S)|$, where $f(\lceil w \rceil_S)$ is the* parent *of $\lceil w \rceil_S$ in T.*

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

4.2 Construction of DBG_k^+

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

Case 1 : $|v| = k$, (Figure 5a)
As $v \in InitExact_{S,k}$, $sl(v) \in SubInit_{S,k}$. Therefore, each child u of $sl(v)$ is an element of $Init_{S,k}$. 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 $RC_S(v) \cap \Sigma$, 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 $Init_{S,k}$ between the root and $sl(u')$, *i.e.* $\lceil first_k(sl(u')) \rceil_S$.

Case 2 : $|v| > k$, (Figures 5b and 5c)
We have that $sl(v)$ is a node of V_T. As $|v| > k$, $|sl(v)| \geq k$. Thus, there exists an element of $Init_{S,k}$ between the root and $sl(v)$. We associate v with this node, *i.e.* $\lceil first_k(sl(v)) \rceil_S$.

We illustrate these two cases in Figure 4:

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 $Init_{S,k}$.

Case 2 Case where v is 1, $sl(v)$ is 2, and $\lceil first_k(sl(v)) \rceil_S$ 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 $u \notin Init_{S,k}$ such that $|u| > k$, one has $p(u) := \lceil first_k(u) \rceil_S$.

The algorithm to build the DBG_k^+ is as follows. A first depth first traversal of T allows to collect the nodes of $Init_{S,k}$ and for each such node to set the pointer $p(.)$ of all its descendants in the tree. Finally to build E^+, one scans

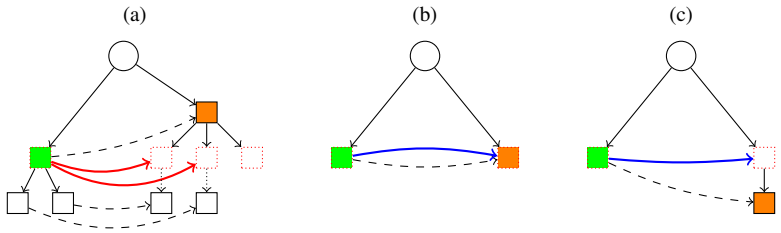

Figure 5: The figures (a), (b) and (c) show the Case 1 and Case 2 to build the arcs of DBG_k^+. The green node is the node v and the orange node is $sl(v)$. The dashed arc corresponds to suffix link and is coloured by specific colour (red (a) for the Case 1 and blue (b) (c) for the Case 2) and in solid line.

through $Init_{S,k}$ and for each node v one adds Succ(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 10. *For a set of words S, building the de Bruijn Graph of order k, DBG_k^+ takes linear time and space in $|T|$ or in $\|S\|$.*

4.3 Construction of CDBG$_k^+$

In Section 3, we have seen an algorithm (in Appendix p. 13) that allows to compute directly CDBG$_k^+$ provided that one can determine if a node v is right extensible and if next(v) is left extensible, where 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 tree, for an element v of $Init_{S,k}$, $first_k(v)$ 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 $\sharp(Support_S(.))$ on the tree, and then how to apply Proposition 7.

Proposition 11. *Let v be a word of F_S and $V_T(\lceil v \rceil_S)$ denotes the set of nodes of the subtree rooted in $\lceil v \rceil_S$.*

$$Support_S(v) = \bigcup_{v' \in V_T(\lceil v \rceil_S)} Suff_S(v').$$

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

Let v be a word of $Init_{S,k}$ 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, $|u| > k$, $\sharp(RC_S(v) \cap \Sigma) = \{u[k+1]\}$, and $last_{k-1}(v)u[k+1] = first_k(sl(u))$. Hence,

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

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

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

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

As $\sharp(Support_S(first_k(v))) = \sharp(Support_S(\lceil first_k(v) \rceil_S))$ and $\sharp(Support_S(v) \setminus \{(i,1) \mid 1 \leq i \leq n\}) = \sharp(Support_S(v)) - \sharp(Support_S(v) \cap \{(i,1) \mid 1 \leq i \leq n\})$, determining if next$(v)$ is left extensible takes constant time. To conclude, as for any initial node v, we can compute in $O(1)$ its set of successors Succ(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 12.

Theorem 12. *For a set of words S, building the Contracted de Bruijn Graph of order k, CDBG$_k^+$ takes linear time and space in $|T|$ or in $\|S\|$.*

5 dBG and CdBG from Suffix Array

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

- $\forall 1 \leq i < \|S\|$, $SA[i] = (g,h)$, $SA[i+1] = (g',h')$ then $s_g[h..|s_g|] < s_{g'}[h'..|s_{g'}|]$,
- $\forall 2 \leq i \leq \|S\|$, $LCP[i]$ 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 5.1 ([10]). *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* & *NSV*) arrays.

Definition 5.2 ([10]). *For $2 \leq i \leq \|S\|$:*

- $PSV[i] = \max\{j \mid 1 \leq j < i$ *and* $LCP[j] < LCP[i]\}$,
- $NSV[i] = \min\{j \mid i < j \leq \|S\|+1$ *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 [10, 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:

- $Init_{S,k}$ 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 $s_g[h..|s_g|]$);
- $InitExact_{S,k}$ is composed of the lcp-intervals k-$[i,j]$;
- $SubInit_{S,k} = InitExact_{S,k-1}$.

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-$[i,j] \in Init_{S,k}$ we have $\sharp(Support_S(s_{SA[i].g}[SA[i].h\,..\,SA[i].h+k-1])) = j-i+1$.

Theorem 13. *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.*

6 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 order 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 [13]. Other tools also exploit this idea [2]. It is thus interesting to dynamically change the order of the dBG. Here, we argue[1] that starting the construction from an index instead of the raw sequences ease the update. On page 7, we mention which information are needed in general to build DBG^+_k. Assume the words are indexed in a suffix tree T (as in Section 4.2). Consider first changing k to $k-1$. First, only the nodes of $Init_{S,k}$ 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 $Init_{S,k-1}$ or $InitExact_{S,k-1}$), and can be computed in constant time.

The same situation arises when changing k to $k+1$. First, only nodes of $InitExact_{S,k}$ 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 children of the source to one children of the destination if both nodes change. Then, the label of the children in T indicate 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 4.3).

7 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 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 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 [2, 13]. Other topics for future research include transforming compressed indexes, such as a FM-index [10], into a dBG, implementing a practical contracted dBG representation for DNA taking into account k-mers and their reverse complements based on these algorithms.

[1] A more formal presentation is left for a complete article.

References

[1] A. Apostolico. The myriad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer, 1985. 2

[2] 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, and P. A. Pevzner. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012. 11

[3] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. In *WABI*, volume 7534 of *LNCS*, pages 225–235, 2012. 1

[4] R. Chikhi, A. Limasset, S. Jackman, J. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. *ArXiv e-prints*, Jan. 2014. 1

[5] R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8:22, 2013. 1

[6] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011. 1

[7] N. de Bruijn. On bases for the set of integers. *Publ. Math. Debrecen*, 1:232–242, 1950. 3

[8] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, 1997. 2

[9] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. 2

[10] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. 604 p. 10, 11

[11] T. Onodera, K. Sadakane, and T. Shibuya. Detecting superbubbles in assembly graphs. In A. Darling and J. Stoye, editors, *Algorithms in Bioinformatics*, volume 8126 of *LNCS*, pages 338–348. Springer, 2013. 2

[12] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. Tiedje, and C. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl Acad. Sci. USA*, 109(33):13272–13277, 2012. 1

[13] Y. Peng, H. Leung, S. Yiu, and F. Chin. IDBA A Practical Iterative de Bruijn Graph De Novo Assembler. In B. Berger, editor, *Research in Computational Molecular Biology*, volume 6044 of *LNCS*, pages 426–440. 2010. 11

[14] P. Pevzner, H. Tang, and M. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, 98(17):9748–9753, 2001. 1

[15] E. A. Rødland. Compact representation of k-mer de Bruijn graphs for genome read assembly. *BMC Bioinformatics*, 14:313, 2013. 1, 2

[16] L. Salmela. Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10):1284–1290, 2010. 2

[17] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010. 2

8 Appendix: a general algorithm for building CDBG$_k^+$

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.

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 intial 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.

```
1  begin
2      u := v_c; mark u
3      // search the node ending the chain that goes through v_c
4      while u is right extensible and next(u) is left extensible do
5          if v_f = next(u) then
6              update (v_f, i) by (v_c, i) for all (v_f, i) ∈ E
7              return (V \ {v_f}, E)
8          u := next(u); mark u
9      // now explore the path starting in the successor of u
10     for w ∈ Succ(u) do
11         if w ∈ V then (V,E) := (V, E ∪ {(v_c, w)})
12         else
13             (V,E) := BuildAuxGdB(V ∪ {w}, E ∪ {(v_c, w)}, v_f, w)
14             // explore from w
15     return (V,E)
16 end
```

Algorithm 2: BuildCDBG(S).

Input : A set of word S.
Output: CDBG$_k^+$ of S.

```
1  begin
2      (V,E) = (∅, ∅)
3      // search for any node v of DBG_k^+ without predecessors
4      // and build CDBG_k^+ from v
5      for v ∈ Init_{S,k} do
6          if there exists no w such that v ∈ Succ(w) then
7              (V,E) := (V,E) ⋃ BuildAuxCDBG(V ∪ {v}, E, v, v)
8      // explore DBG_k^+ from any node not yet visited
9      for v_c an unmarked node of Init_{S,k} do
10         (V,E) := (V,E) ⋃ BuildAuxCDBG(V ∪ {v_c}, E, v_c, v_c)
11     return (V,E)
12 end
```

A more detailed explanation. First, note that to build DBG_k^+ it suffices to know the set Succ$(.)$ 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 $Init_{S,k}$, we traverse the simple path until we arrives 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 represents 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 a simple 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 belong 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.