Construction of a de Bruijn Graph for Assembly from a Truncated Suffix Tree

Bastien Cazaux, Thierry Lecroq, Eric Rivals

Construction of a de Bruijn Graph for Assembly from a Truncated Suffix Tree*

Bastien Cazaux[1], Thierry Lecroq[2], and Eric Rivals[1]

[1] L.I.R.M.M. & Institut Biologie Computationnelle, Université de Montpellier II, CNRS U.M.R. 5506, Montpellier, France
[2] LITIS EA 4108, NormaStic CNRS FR 3638, Université de Rouen, France
cazaux@lirmm.fr, thierry.lecroq@univ-rouen.fr, rivals@lirmm.fr

Abstract. In the life sciences, determining the sequence of bio-molecules is essential step towards the understanding of their functions and interactions inside an organism. Powerful technologies allows to get huge quantities of short sequencing reads that need to be assemble to infer the complete target sequence. These constraints favour the use of a version de Bruijn Graph (DBG) dedicated to assembly. The de Bruijn Graph is usually built directly from the reads, which is time and space consuming. Given a set R of input words, well-known data structures, like the generalised suffix tree, can index all the substrings of words in R. In the context of DBG assembly, only substrings of length $k+1$ and some of length k are useful. A truncated version of the suffix tree can index those efficiently. As indexes are exploited for numerous purposes in bioinformatics, as read cleaning, filtering, or even analysis, it is important to enable the community to reuse an existing index to build the DBG directly from it. In an earlier work we provided the first algorithms when starting from a suffix tree or suffix array. Here, we exhibit an algorithm that exploits a reduced version of the truncated suffix tree and computes the DBG from it. Importantly, a variation of this algorithm is also shown to compute the contracted DBG, which offers great benefits in practice. Both algorithms are linear in time and space in the size of the output.

Keywords: Stringology, Text Algorithms, Indexing Data Structures, de Bruijn Graph, Assembly, Space Complexity, Dynamic Update

1 Introduction

The de Bruijn Graph (DBG) serves in bioinformatics and genomics to assemble the sequence of large molecules from a huge set of short sequencing reads. In this context, only the substrings of length, say k, of the reads form the nodes of the DBG (unlike in the original DBG). These substrings are termed k-mers in biology or k-grams in computer science. An arc links two k-mers whenever they overlap by $(k-1)$ symbols at (necessarily) successive positions in a read. The assembly DBG is then traversed searching for long paths, which will form the *contigs*, i.e. the sequence of sub-regions of

* This work is supported by ANR Colib'read (http://colibread.inria.fr) (ANR-12-BS02-0008) and by Défi MASTODONS SePhHaDe (http://www.lirmm.fr/mastodons) from CNRS.

the molecule. However, in non repetitive regions, the layout of the reads dictate a single 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 sequence of such simple paths are called *unitigs* (the contraction from unique and contigs). The version of the DBG where all such "non-branching" paths are condensed into an arc is termed the Contracted DBG (CDBG).

Given the extreme throughput delivered by nowadays sequencing machines, it is crucial to enable fast construction of the CDBG. It is also desirable to build the DBG or its contracted version directly from space efficient indexing data structures, since the read set has often been filtered and mined for patterns representing errors prior to assembly using such data structures. It occurs for instance in a preprocessing phase of sequencing errors removal using a *generalised suffix tree* of the reads [15]. The use of an indexing data structure (or index for short) also allows to compute and store additional information into the nodes of the DBG: the coverage of a k-mer, i.e. the number of reads in the layout covering that k-mer. Unexpected variations of the coverage permits to detect sequencing errors, to distinguish between classes of point mutations [13], but also to disentangle repetitive sequence regions. Indeed, the reads coming from the distinct but similar copies of a repeat tend to collapse into a single assembly region and increase abnormally the local coverage. Actually, the DBG is itself used as a data structure to seek graph patterns representing mutation, large insertions/deletions, or chromosomal rearrangements [14].

In a first attempt towards constructing the DBG from an index, Cazaux *et al.* gave recently two algorithms for building it from either a Generalised Suffix Tree (GST) or a generalised Suffix Array [4]. Indeed, a subset of the suffix tree nodes represent either exactly one k-mer or its shortest extension; hence, this subset is isomorphic to the set of nodes of the DBG. Moreover, following a suffix link and then going down the tree at most once allows to traverse from one node to its neighbours in the DBG. Hence, the arcs of the DBG can be simulated on the GST or computed using it. This summarises the basis of the DBG construction algorithms, which require linear time in the input length (i.e. the cumulated sum of the read lengths). Importantly, it was also shown that the contraction of the arcs in the DBG can be computed in linear time during the construction: this gave the first linear time CDBG construction algorithm [4].

However, in practice the size of the Generalised Suffix Tree remains prohibitive for large read sets. In the light of these algorithms, it is clear that many nodes of the GST among those having a string depth either larger than the order k or strictly smaller than $k-1$ are useless. Truncated Suffix Trees (TST) have been introduced to index only a subset of factors below a certain string depth [10, 16]. TST avoid storing strings longer than a limit length k, but still include all suffixes shorter than k. In practice, when the reads are numerous and short, which occurs in a majority of large sequencing applications, the memory wasted for such nodes is important. With an Illumina sequencing experiment, the typical number of reads $n := 10^8$, the read length is 100 and several values are used for k up to 64. In such a case, one stores $n \times (k-2)$ useless nodes. We set out to first find an algorithm for a reduced version of the TST that avoids those nodes, second to show that the DBG and CDBG can both be built in time and space that are linear in the

size of the DBG, rather than in the cumulated length of the reads. The paper is organised as follows. Below we list related works. In Section 2, 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 [9] for doing so. In Section 3, we introduce the reduced truncated suffix tree and specialise the previous algorithm for constructing it efficiently. Finally, in Section 4 we show how to construct both the de Bruijn Graph and its contracted version in optimal time from the reduced truncated suffix tree. We then conclude mentioning lines of future work.

1.1 Related Works

Suffix trees are well-known indexing data structures that enable 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 (GST). They can be built in linear time and space. They have been widely studied and used in a large number of applications (see [1, 8]). In some applications it is not required to consider the full set of suffixes, since one may only be interested in factors of length bounded by a given constant. These factors are actually prefixes of suffixes. In 2003, Na *et al.* [10] introduced the truncated suffix trees which only stores the factors of length at most k in a context of lossless data compression. They gave linear time algorithms for directly constructing truncated suffix trees. They present experimental results showing that on various kinds of strings and different values of k that truncated suffix trees have much less nodes than suffix trees. Truncated suffix trees have been generalised to set of strings and use for performing efficient pattern matching in biological sequences [16].

DBGs are heavily exploited for genome assembly in bioinformatics [12], where several compact data structures for storing DBGs have been developed [6, 2] including probabilistic ones [5]. 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 [11], thereby emphasising the need for dynamically updating the DBGs.

From now on, the input of our problem consists of an integer $k > 0$ and $R = \{w_1, \ldots, w_n\}$ a set of n finite words over a finite alphabet. We will consider indexing the substrings of words in R; hence, all indexes used are *generalised* indexes [8]. For simplicity, we may omit this adjective.

2 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 [9] and the truncated suffix tree [10]. First, we introduce a notation on strings.

Notation on Strings We consider finite strings (also termed words or sequences) over a finite alphabet Σ. For a string w, $|w|$ denotes the *length* of w. For any integers i and j such that $i \leq j$, $[i, j]$ denotes the interval of integers between i and j. For any $i \leq j$

in $[1,|w|]$, $w[i,j]$ is the substring of w beginning at position i and ending at position j. Then, $w[1,i]$ is called a *prefix* of w, while $w[i,|w|]$ is a *suffix* of w. For a set of strings A, the *norm* of A, denoted $||A||$, is

$$||A|| = \sum_{a \in A} |a|.$$

Now, let us define the concept of suffix-dependant strings and of chains of suffix-dependant strings.

Definition 1. *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 $|x_i| \geq |x_{i-1}|-1$.*

For a set of strings $R = \{w_1,\ldots,w_n\}$, let $\mathcal{S} = \{C_1,\ldots,C_n\}$ 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 w_i. For $i \in [1,n]$ and $j \in [1,|C_i|]$, $C_i[j]$ is the j^{th} string of the tuple C_i. Let be $\widehat{\mathcal{S}} = \{\widehat{C_1},\ldots,\widehat{C_n}\}$ 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. $\widehat{\mathcal{S}}$ contains tuples of lengths.

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

Let w be a string; w may occur in distinct tuples of $\mathcal{S}$. 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 $\mathcal{S}$ that are equal to w.

We define a contracted version of the well-known Aho-Corasick tree [8]. In fact, we apply nearly the same contraction process that turns a trie of a word into its compact Suffix Tree [8]. Consider the Aho-Corasick tree of R, in which each node represents a prefix of word in R. We contract the non-branching parts of the branches except that we keep all nodes representing a word that belongs to a tuple in $\mathcal{S}$. From now on, let $T(\mathcal{S})$ denote this contracted version of the Aho-Corasick tree of R.

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

- $s(v)$ is the number of times that an element of a tuple of $\mathcal{S}$ 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 $\mathcal{S}$ 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 $\mathcal{F}$ as the subset of $\mathcal{L}$ such that:

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

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

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

Algorithm to Construct the $T(\mathcal{S})$: Now, we give an algorithm to construct $T(\mathcal{S})$. We use the version of McCreight's algorithm given by Na et al. [10] on our input and we build for each leaf v, $s(v)$, $t(v)$ and $P_0(v)$. For building $T(\mathcal{S})$, 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 $T(\mathcal{S})$.

Theorem 2. *For a set of chain of suffix-dependant strings $\mathcal{S}$, we can construct $T(\mathcal{S})$ in $O(||R||)$ time and space.*

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

Clearly, the construction algorithm of $T(\mathcal{S})$ 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 $\mathcal{S}$, 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 $O(||R||)$.

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

- If $C := \cup_{w\in R}\{\text{tuple of suffixes of } w\}$, then $T(C)$ is the Generalised Suffix Tree of R (see Figure 1a). We have that the restrained mapping $sl(.)$ is an example of a possible link.
- If $B_k := \cup_{w\in R}\{\text{tuple of k-mer of } w \text{ and suffixes of length } k' < k \text{ of } w\}$, then $T(B_k)$ is the generalised k-truncated suffix tree of R, as defined in [16] (which generalizes the k-truncated suffix tree of Na et al. [10]).
- If $A_k := \cup_{w\in R}\{\text{tuple of } k+1\text{-mer of } w \text{ and suffixes of length } k \text{ of } w\}$, then $T(A_k)$ is the truncated suffix tree that we define below in Section 3 (see Figure 1b).

3 Our Truncated Suffix Tree

First, let us introduce a notation about trees that index strings and whose edges are labelled with strings.

Definition 3. *For a node v of a tree, $f(v)$ denotes the parent node of v, and $Children(v)$ its set of children. The depth of v is the length of the unique path between the root and v in the tree. Each represent a unique word: that made up by the concatenation of the label*

of edges along this path. The notion of node and the word it represents are confounded (we used one for the other). For a $T(\mathcal{S})$ and a word w, $\lceil w \rceil$ is the node v with the shortest depth of the $T(\mathcal{S})$ such that w is a prefix of v. If this node does not exist, $\lceil w \rceil$ does not exists. For a node u of the $T(\mathcal{S})$, $sl(u)$ is the node $\lceil u[2,|u|] \rceil$.

For a set of words $R = \{w_1, w_2, \ldots, w_n\}$ and an integer $k > 0$, we define the following notation.

Definition 4.

1. *$F_k(R)$ is the set of substrings of length k of words of R.*
2. *$Suff_k(R)$ is the set of suffixes of length k of words of R.*
3. *For all $i \in [1,|R|]$ and $j \in [1,|w_i|-k+1]$, $A_{k,i}$ denotes the tuple such that its j^{th} is defined by*

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

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

Proposition 5. *1. $A_{k,i}$ is a chain of suffix-dependant strings of w_i.*
2. Moreover, $\{w \in A_{k,i} \mid A_{k,i} \in A_k\} = F_{k+1}(R) \cup Suff_k(R)$.

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

$$\begin{aligned} \{w \in A_{k,i} \mid A_{k,i} \in A_k\} &= \bigcup_{i=1}^{n} \Big(\bigcup_{j=1}^{|w_i|-k+1} \{A_{k,i}[j]\} \Big) \\ &= \bigcup_{i=1}^{n} \Big(\bigcup_{j=1}^{|w_i|-k} \{A_{k,i}[j]\} \bigcup \{A_{k,i}[|w_i|-k+1]\} \Big) \\ &= \bigcup_{i=1}^{n} (F_{k+1}(\{w_i\}) \bigcup Suff_k(\{w_i\})) \\ &= F_{k+1}(R) \cup Suff_k(R) \end{aligned}$$

By applying the algorithm described in Section 2 to the set A_k (Definition 4), and by using Theorem 2, we get the following result.

Corollary 6. *We can construct $T(A_k)$ in $O(||R||)$ time and space.*

3.1 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

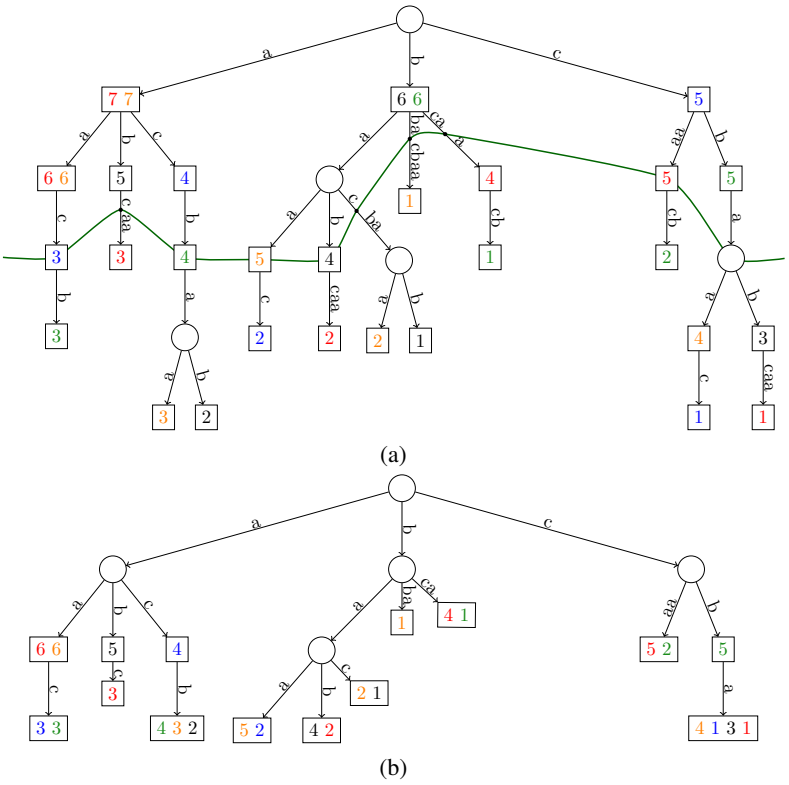

Fig. 1. (a) The generalised suffix tree for the set of words {*bacbab*, *bbacbaa*, *bcaacb*, *cbaac*, *cbabcaa*}. The part above the green line corresponds to the TST $T(A_2)$, which is shown in (b). (b) The truncated suffix tree $T(A_2)$ for the same set of words.

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 [16]. It should be noted that their implementation of the TST stores all the suffixes shorter than k producing thus more nodes than our TST. Table 1 show the results. It can be seen that for small sets, TSTs do not save many nodes compare 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 De Bruijn Graph via the Truncated Suffix Tree

Here, we describe an algorithm that builds the de Bruijn Graph of a set of words R starting from the generalised truncated suffix tree of R. Note that this DBG differs from the original graph as defined by de Bruijn in the field of word combinatorics [3]. The

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

Table 1. Number of nodes of the GST vs the TST for $k = 5, 10, 20, 40$ and the percentage compare to the GST for Illumina reads of length 101.

♯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%)

DBG studied here serves for genome assembly and for approximating the well-known Shortest Superstring problem [7].

4.1 De Bruijn Graph

Let k be a positive integer and $R := \{w_1, \ldots, w_n\}$ be a set of n words. We use the definition of a de Bruijn graph stated in [4].

Definition 7 (de Bruijn graph). *The* de Bruijn graph *of order k for R, denoted by dBG_k^+, is a directed graph, 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 R.*

Another slightly different definition is sometimes used in which the k-mers must overlap by $(k-1)$ symbols, but must not necessarily occur in the same read. This relaxed definition introduces "false" arcs, but is easier to build and store in memory. All our results can easily be adapted to that definition.

Proposition 8 states that there does not exist any leaf in $T(A_k)$ representing a word strictly shorter than k.

Proposition 8. *Let v be a leaf of $T(A_k)$. Then $|v| = k$ or $|v| = k+1$.*

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

We set $Init_{R,k} = \{v \in V_{T(A_k)} \mid |v| \geq k \text{ and } |f(v)| < k\}$. For a possible link P_0, we define the mapping P from $\mathcal{F}$ to $\mathcal{N}$. $\mathcal{F}$, $\mathcal{N}$ and $\mathcal{L}$ have the same definition as before, but applied to the $T(A_k)$. $\mathcal{F}$ 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:

$$P : \mathcal{F} \longrightarrow \mathcal{N}$$
$$v \mapsto \begin{cases} P_0(v) & \text{if } P_0(v) \in Init_{R,k} \\ f(P_0(v)) & \text{otherwise} \end{cases}$$

The mapping P can be constructed in linear time in $O(||R||)$. In fact, for each $v \in \mathcal{F}$, $P(v)$ can be constructed in $O(1)$ because in this case, $P_0(v) \in Init_{R,k} \Leftrightarrow |f(P_0(v))| \neq k$.

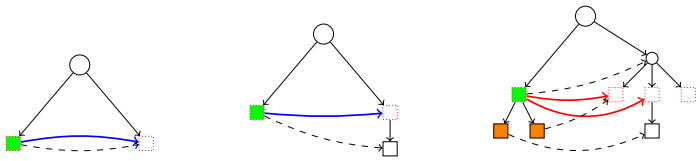

Fig. 2. The different cases of the definition of E^+. For a node v (in green), the curved solid arrows are the arcs of E^+, the dashed arrows are the suffix links of the nodes, the dashed nodes are nodes of $Init_{R,k}$. (a) and (b) correspond to the first part of the definition while (c) corresponds to the second part.

As $|\mathcal{F}| \leq ||R||$, we can construct P for all elements of $\mathcal{F}$ in $O(||R||)$. Indeed, It is enough to look the length of the parent of $P_0(v)$ to decide if $P_0(v)$ is in $Init_{R,k}$.

Proposition 9. *Le be $v \in \mathcal{L}$, $P(v) \in Init_{R,k}$ and $P(v) = sl(v)$ if $sl(v)$ exists.*

Proof. Let be $v \in \mathcal{L}$. If $v \in \mathcal{F}$ and $P_0(v) \notin Init_{R,k}$, $|f(P_0(v))| = k$ and thus $P(v) = f(P_0(v)) \in Init_{R,k}$. 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, $P(v) = sl(v)$.

Let us define, E_k^+, a set of arcs. Two cases arise depending on whether the starting node v represents a word of length k or $k+1$.

$$E_k^+ = \left(\cup_{|v|=k+1}(v, P(v))\right) \cup \left(\cup_{|v|=k}\left(\cup_{u \in Children(v)}(v, P(u))\right)\right)$$

with $v \in Init_{R,k}$. Figure 2 illustrates the alternative cases in the definition of E_k^+, which is the union of these cases.

Proposition 10. *$(Init_{R,k}, E_k^+)$ is isomorphic to dBG_k^+ of R.*

Proof. The proof is identical to that of [4, Theorem 1].

Figure 3 shows an example of de Bruijn graph of order 2 built from $T(A_2)$.

Proposition 11. *The size of $T(A_k)$ is linear in the size of dBG_k^+.*

Proof. Let b be the application from $\mathcal{F}$ to E_k^+ such that

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

By Proposition 10, b is a bijection. As $|\mathcal{L} \setminus \mathcal{F}| \leq |R|$, $|\mathcal{L}|$ is linear in the size of dBG_k^+.

From R, we first build the reduced TST, and then build dBG_k^+ from it. By Proposition 11, we get the following theorem.

Theorem 12. *For a set of words R, we can construct dBG_k^+ in $O(||R||)$ time and in $O(size(dBG_k^+))$ space.*

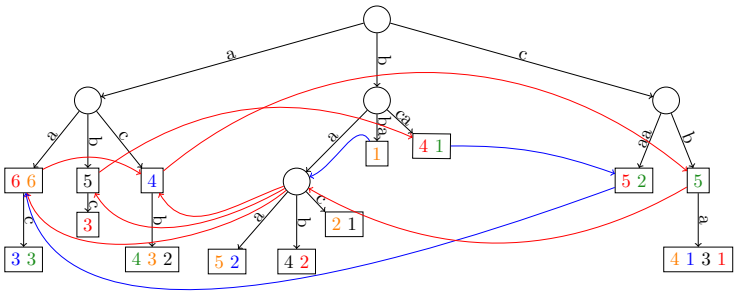

Fig. 3. The de Bruijn graph of order 2 built on $T(A_2)$. The solid curved arrows are the edges corresponding to the first part of the definition of E_k^+, while those in blue correspond to the second part.

4.2 A Contracted de Bruijn Graph

Let e an arc of dBG_k^+, $e = (x,y)$ is said to be *reducible* if the outdegree of x and the indegree of y are at most 1.

Definition 13. *Let k be a positive integer and $R := \{w_1, \ldots, w_n\}$ be a set of n words. The* contracted de Bruijn graph *of order k for R, denoted by $CdBG_k^+$, is the de Bruijn graph of order k where each reducible arc is contracted.*

Let z be a word, $Support_R(z)$ is the set of pairs (i,j), where z is the substring $w_i[j, j+|s|-1]$. $Support_R(z)$ is called the support of z in R.

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

Hence, we obtain the following theorem.

Theorem 15. *For a set of words R, we can construct $CdBG_k^+$ in $O(||R||)$ 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 R. These leaves make $T(B_{k+1})$ linear in $||R||$, but not in the size of dBG_k^+.

5 Conclusion

First, we provided a unified version of the construction of a generalised index for a set of input words. It only requires that the elements of the chain are suffix-dependant. This general framework was applied to build a reduced version of a Truncated Suffix Tree (TST), which given a parameter k does not contain any node representing a string smaller than $k-1$ compared to the TST of [10, 16]. Our algorithm does not build those nodes

and thus does not need to remove them afterwards, as would be the case with the TST of [10, 16]. This feature is crucial since otherwise, the index data structure could not be linear in the size of the de Bruijn Graph of order k. Moreover, it is not trivial to modify the algorithm of [10] to avoid building those nodes, because it uses their suffix links during construction. A natural question arise: does our framework remain valid if one relaxes the assumption of suffix-dependency?

Second, we show that the de Bruijn Graph (DBG) that is heavily exploited in the context of genome assembly can be directly constructed from the reduced TST in a time and a space linear in the size of the output. This part builds upon our earlier work, which addressed the question of DBG construction from a Suffix Tree and Suffix Array [4]. Moreover, we provide an algorithm to build the contracted DBG, which is used in practice and directly yields as a by-product of the contraction, a class of reliable contigs, called the unitigs[2]. Starting from the reduced truncated tree ensures that the index encodes as much nodes as the output DBG. However, as information on nodes representing substrings of length $< k-1$ are missing, one looses the dynamicity obtained in [4, Section 6]: our reduced TST cannot support the DBG construction for both order k and $k-1$. Nevertheless, as our algorithms remain valid for the original TST [10, 16], which stores all the information for any substrings $\leq k$, we can dynamically update the DBG for any order $< k$. Both the questions of 1/ dynamically updating the de Bruijn Graph when the order changes by more than one unit, and of 2/ constructing the DBG from compressed indexes make thrilling lines of future work.

References

1. Apostolico, A.: The myriad virtues of suffix trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words, NATO Advanced Science Institutes, Series F, vol. 12, pp. 85–96. Springer (1985)
2. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn Graphs. In: Workshop on Algorithms in BioInformatics (WABI). Lecture Notes in Computer Science, vol. 7534, pp. 225–235 (2012)
3. de Bruijn, N.: On bases for the set of integers. Publ. Math. Debr. 1, 232–242 (1950)
4. Cazaux, B., Lecroq, T., Rivals, E.: From Indexing Data Structures to de Bruijn Graphs. In: Kulikov, A., Kuznetsov, S., Pevzner, P. (eds.) Proc. of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM). Lecture Notes in Computer Science, vol. 8486, pp. 89–99 (2014)
5. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology 8, 22 (2013)
6. Conway, T.C., Bromage, A.J.: Succinct data structures for assembling large genomes. Bioinformatics 27(4), 479–486 (2011)
7. Golovnev, A., Kulikov, A., Mihajlin, I.: Approximating shortest superstring problem using de bruijn graphs. In: Fischer, J., Sanders, P. (eds.) Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 7922, pp. 120–129. Springer Berlin Heidelberg (2013)
8. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)

[2] Usually the result of an assembly is not the complete sequence, rather a set of assembled sequences called contigs

9. McCreight, E.: A space-economical suffix tree construction algorithm. J. of Association for Computing Machinery 23(2), 262–272 (April 1976)
10. Na, J.C., Apostolico, A., Iliopoulos, C.S., Park, K.: Truncated suffix trees and their application to data compression. Theoretical Computer Science 1-3(304), 87–101 (2003)
11. Peng, Y., Leung, H., Yiu, S., Chin, F.: IDBA – A Practical Iterative de Bruijn Graph De Novo Assembler. In: Berger, B. (ed.) Research in Computational Molecular Biology, LNCS, vol. 6044, pp. 426–440. Springer Berlin Heidelberg (2010)
12. Pevzner, P., Tang, H., Waterman, M.: An Eulerian path approach to DNA fragment assembly. Proc. Natl Acad. Sci. USA 98(17), 9748–9753 (2001)
13. Philippe, N., Salson, M., Commes, T., Rivals, E.: CRAC: an integrated approach to the analysis of RNA-seq reads. Genome Biology 14(3), R30 (2013)
14. Rizk, G., Gouin, A., Chikhi, R., Lemaitre, C.: Mindthegap: integrated detection and assembly of short and long insertions. Bioinformatics (2014)
15. Salmela, L.: Correction of sequencing errors in a mixed set of reads. Bioinformatics 26(10), 1284–1290 (2010)
16. Schulz, M.H., Bauer, S., Robinson, P.N.: The generalised k-truncated suffix tree for time-and space-efficient searches in multiple DNA or protein sequences. International J. of Bioinformatics Research and Applications 4(1), 81–95 (2008)