Succint de Bruijn Graphs: implementation report

Brilli Matteo

August 22, 2024

Abstract

Following the paper "Succint de Bruijn Graphs" by Bowe et al. (2012) [1], we implement a way to represent de Bruijn graphs of k-mers in a DNA sequence with m edges using $4m + o(m)$ bits, and show how this space can be compressed even further. Using this data structure we can find the outdegree and indegree of a node and the outgoing edge with a given label in constant time. The incoming edge with a given label is found in $\mathcal{O}(k \log \sigma)$ time, while the label of a node and membership queries can instead be answered in $\mathcal{O}(k)$ time.

Introduction

A de Bruijn graph (dBG) is a directed graph, named after the Dutch mathematician Nicolaas Govert de Bruijn, used to represent overlaps between sequences of symbols. In the *node-centric* dBG, the type we will focus our discussion on, the nodes correspond to all the possible sequences of length k formed with symbols from a set Σ; the symbols can be repeated. Each graph will thus have σ^k distinct nodes, each representing a unique sequence of symbols (we use $\sigma = |\Sigma|$ to indicate the alphabet size).

Let's denote each node by its sequence of symbols $(c_1, c_2, \dots, c_k)$. For any pair of nodes $x = (x_1, x_2, \dots, x_k)$, $y = (y_1, y_2, \dots, y_k)$, a directed edge will be drawn from x to y if and only if $x_2 = y_1, x_3 = y_2, \dots x_k = y_{k-1}$. This edge will be labeld with y_k.

For example, given $\Sigma = \{a, b\}$ and $k = 3$, there would be and egde labeled b going from node aab to node abb, since the last $k-1$ characters of aab are equal to the first $k-1$ characters of abb.

Storing a de Bruijn graph requires huge amounts of memory. In 2012, Bowe, Onodera, Sadakane and Shibuya [1] proposed a succinct representation based on the Burrows-Wheeler transform. Their representation, which we will refer to as BOSS from the authors' initials, requires only $(2+\log\sigma)m$ bits to be stored, where m is the number of edges. This size does not depend on the length k.

In this report, we first briefly introduce how and why we use de Bruijn graphs for DNA assembly, then we summarize the main points of the analyzed paper [1] while discussing the details of a custom C++ implementation. We'll finally conclude analyzing the performance of the implementation and the space and time complexities. To do this, we also use as supporting material what's reported in the blog post [2] on the argument written by one of the authors of the paper.

Preliminaries

DNA sequencing

The expression DNA sequencing refers to the process of determining the sequence of nucleotide bases (As, Ts, Cs, and Gs) in a string of DNA. Fast DNA sequencing enables the identification and cataloging of more species as well as the delivery of more tailored and efficient medical treatment. Unfortunately, sequencing an entire genome remains a very complex task.

Some of the most recently developed sequencing techologies are categorized under the umbrella classification of Next-generationg sequencing (NGSs). They use a variety of different techniques, but share a common set of features: they are highly parallel, very fast, and cheaper the previous generation sequencers. Most importantly, they also generate reads that are shorter in lenght, generally ranging from 50 to 900 nucleotides.

Because of this, the fragments generated from a longer DNA sequence have to be subsequently aligned and merged in order to reconstruct the original string. This process is know as DNA assembly.

de Bruijn Grahps

In this context, dBGs are used for de novo assembly of reads. De novo sequence assemblers are a type of program that assembles short nucleotide sequences into longer ones without the use of a reference genome.

During the assembly of the dBG, reads are broken into smaller fragments of a specified size k. Each node in the graph represents a k-mer (a substring of length k) that exists in the reads, and an edge exists if and only if there is an exact overlap of length $k-1$ between the corresponding k-mers. The label of each node therefore can be reconstructed by walking along the k predecessor edges.

The concept of a dBG is originally borrowed from combinatorics. In bioinformatics, we actually work with subgraphs of the complete dBG, i.e. composed of the whole set of σ^k nodes. Infact, even though genomes are long strings, most genomes won't have every single k-mer present, and there is usually repeated regions.

Succinct representation

As we already mentioned, the BOSS data structure proposes a novel exact dBG representation using a variant of the Burrows-Wheeler transform specifically tailored for k-mers. Intuitively, the Burrows-Wheeler transform is a permutation of the characters of a string that facilitates substring search and compression. BOSS extends this concept by storing a permutation of the last characters of each k-mer together with a bit array. The result is a data structure that supports efficient membership queries and neighborhood traversal of the graph.

Following [1], we'll show how to construct the k-dimensional de Bruijn graph G of a string T of length N on alphabet Σ. The nodes of the graph will correspond to all length-k substrings of T, so the graph will have at most $N - k + 1$ nodes. For convenience, we also add k terminator characters \$ at the head of the string, and a \$ at the end.

We can also store a set of M strings $T_1, \dots, T_M$ by appending a terminator $\$_i$ to the tail of each string T_i, and concatenating all the strings. Then we add k characters $\$_0$ at the head. In figure (1) is shown an example for the strings T_1 ="TACAC", T_2 ="TACTC", T_3 ="GACTC". As a result, the final string from which the de Bruijn graph is constructed is T ="$\$_0\$_0\$_0$TACAC$\$_1$TACTC$\$_2$GACTC$\$_3$".

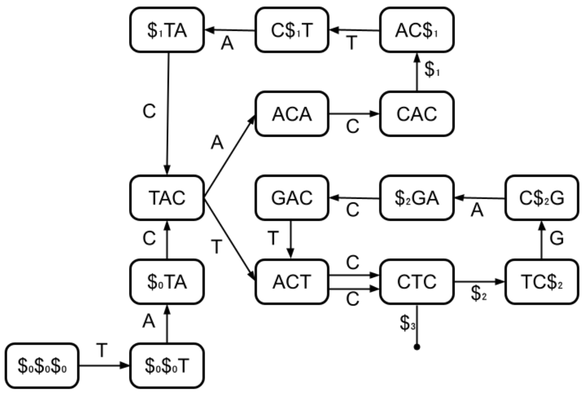


Figure 1. The 3-dimensional de Bruijn graph of strings 'TACAC', 'TACTC', and 'GACTC'.

During construction we will actually unify the M terminators $\$_1, \dots, \$_M$ into a single character \$. This makes the practical implementation feasible while also reducing the alphabet size.

Let n and m be the numbers of nodes and edges of G, respectively. We will indicate a node via an index $v \in [0, n-1]$, while an edge via $i \in [0, m-1]$. The proposed succinct representation of G supports the following operations:

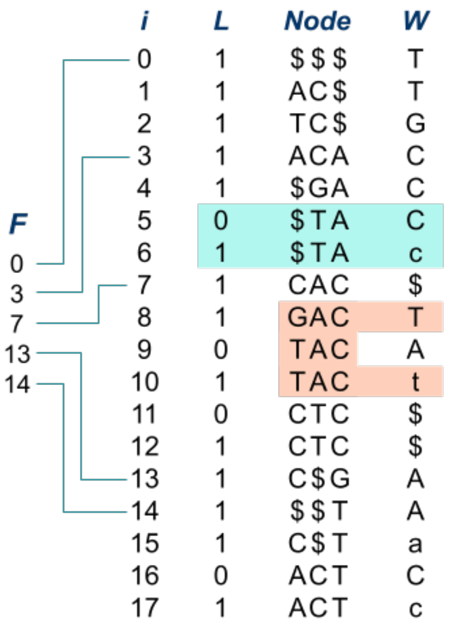
- `Outdegree(v)` returns the number of outgoing edges from node v.
- `Outgoing(v, c)` returns the node w pointed to by the outgoing edge of node v with edge label c. If no such node exists, it returns -1.
- `Indegree(v)` returns the number of incoming edges to node v.
- `Incoming(v, c)` returns the node $w = (w_1, \dots, w_k)$ such that there is an edge from w to v and $w_1 = c$. If no such node exists, it returns -1.
- `Index(s)` returns the index i of the node whose label is the string s of length k. If such node doesn't exists it returns -1.
- `Label(v)` returns the label of node v.

Theoretical construction

The first step in the construction of the k-dimensional de Bruijn graph of a string T consists in taking every (node, edge) pair and sorting them in colex order, i.e. based on the reverse of the node label. Then, the proposed representation is composed of the following three elements;

- a string W of length m over alphabet Σ;
- a bitvector L of length m;
- an array F of length σ.

An example is shown in Figure (2).



F	*i*	*L*	*Node*	*W*
	0	1	\$\$\$	T
	1	1	AC\$	T
	2	1	TC\$	G
	3	1	ACA	C
	4	1	\$GA	C
	5	0	\$TA	C
0	6	1	\$TA	c
3	7	1	CAC	\$
7	8	1	GAC	T
13	9	0	TAC	A
14	10	1	TAC	t
	11	0	CTC	\$
	12	1	CTC	\$
	13	1	C\$G	A
	14	1	\$\$T	A
	15	1	C\$T	a
	16	0	ACT	C
	17	1	ACT	c

Figure 2. Example for T="\$\$\$TACAC\$TACTC\$GACTC\$"

Each character in W represents the label of an edge of the graph. Each edge is associated which the node from with it exits. Let's denote with $node[i]$ the label of the node associated with $W[i]$. Then, for example, we have the edge $W[1]$ ="T" exiting from the node labelled $node[1]$ ="AC\$" and pointing to $node[15]$ ="C\$$T$". Note that only W is explicitely stored, not the labels of the nodes.

Some of the edges in Figure (2) are represented with lowercase letters. This is needed to distinguish between identically labelled incoming edges, i.e. edges that enter the same node even though they exit from different ones. As an example, the edges highlighted in apricot in Figure (2), that is $node[8]$ and $node[10]$, both point to the node with label "ACT", even though they exit from different nodes. Although in the paper [1] this kind of edges are flagged with a minus sign, we found it easier on a practical level to use lowercase characters. This is because this change makes it possible to provide rank and select support on strings (all details will be explained in the following sections), while still mantaining the same space complexity as in the paper.

Thanks to the colex order each outgoing edge exiting from the same node is stored contiguously. Therefore, we include a bit vector to represent whether an edge is the last edge exiting a node. L is defined as $L[i] = 1$ if $i = n$ or $node[i]$ is different from $node[i+1]$. From this definition, all node labels with $L[i] = 1$ are distinct, and those indices i have one-to-one correspondence with the nodes of G. Thus there will be a total of n ones in the bitvector L. Since a 1 in the vector identifies a unique node, we can use this vector (and select, explained shortly) to index nodes, whereas standard array indexing points to edges.

Finally, we also need to know the final column of the node labels. Since they are sorted, we can simply store the cumulative frequencies in the array F.

In total we have a bitvector L, a string of flagged edge labels W, and an array of integers F of size σ. Respectively, these take m bits, $m \log(2\sigma + 1)$ bits, and $\sigma \log m = o(m)$ bits, i.e. a bit over 4 bits per edge (for DNA). We'll see how to compress this further by using appropriate data structures in the next sections.

Handling DNA repeats

When trying to reconstruct the genome we not only need to find all occuring k-mers, but also find how many times each such k-mer appears, i.e. its 'multiplicity'. In the following implementation, we simply follow the construction method just described, without discriminating repeated edges. This means we'll store repeated entries, which will end up next to each other, like the couple highlighted in celeste if Figure (2). The good news is that the graph resulting from adding multiplicity edges is balanced, i.e. the indegree and outdegree of each node are equal, and they correspond to the number of times this k-mer appears in the genome. Another way of keeping track of multeplicities would be to store an array of the weights corresponding to each node.

■ Implementation

Data input

Let's start to discuss the details of the implementation by briefly talking about the format of the input data. The presented approach can be easily adapted to any type of input file, but we'll imagine to work with FASTQ files. FASTQ format is a human-readable file format that stores the nucleotide base sequences, the calculated confidence for each base in a sequence, and information describing the origin of the read. All FASTQ files consist of a set of reads, with each read having 4 lines of data, of which the second line is the one that contains the nucleotide base sequence.

The present implementation needs as input the path to a file that contains a single line created concatenating the reads as we have described previously. It is important though to reverse this string in place, for a reason that will become clear very shortly.

Thus, we need a first preprocessing stage, in which we take the different nucleotide sequences, reverse them in place, append a $ to the end and concatenate them. Finally, we need to append the k terminator characters. Continuing with the same example, the file should contain the string "CTCAG$CTCAT$CACAT$$$". Note that the terminator character that was previously marked with $\$_3$, and that here should be in the first position, must not be present, as it will be automatically taken into consideration during the first step of construction.

Class construction

In the implementation, the data structure is implemented through a C++ class called `BOSS`. In this section we'll go over the code for the constructor of the class, and afterwards we'll examine the various functions needed for graph traversal (i.e. `outgoing`, `outcoming`,...). This implementation depends on the use of the Succinct Data Structures Library (SDSL) [3], which is used to store in compressed form the various members of the class.

We have said that the first step of the construction algorithm consists in taking all the different (node, edge) pairs and sorting them in colex order. The corresponding first part of the code will be based on the creation of a Compressed Suffix Array (CSA) using the SDSL library. This will be useful to obtain most of the informations we need.

```
BOSS(const std::string& file_path, int k) {
    sdsl::csa_wt<> csa;
    sdsl::construct(csa, file_path, 1);
    this->m = csa.size() - k;

    ...
```

Code 1. CSA construction

The `sdsl::construct` method builds the CSA directly from a file, so we have to pass as argument the path to the file we built in the previous section. The CSA object can now be used to access all suffixes of the string in alphabetical order through `csa[i]`. Since we need the node labels to be sorted in colex order, it now becomes clear why we reversed the input string in place during preprocessing. Note that we must discard the first k elements of the CSA, since they are generated only because of the first k terminator characters $\$_0$ we added at the beginning, and are not needed for the BOSS data structure.

The construction of the CSA actually serves multiple purposes: it immediately gives us W, which is exactly `csa.bwt`, but also F, which is accessible through `csa.C`. In both cases we only have to take the slight precaution of eliminating the info about the first k elements. What we need to do now is to create the bitvector L and flag the elements of W, using a for loop that runs over all k-mers in colex order. As temporary storage for the duration of the for loop we need to use a basic `sdsl::bitvector` for L and a string for W, but we'll compress them later.

The first step of the for loop consists in the extraction of the node label. We will need to consider the first k characters of every suffix, and reverse them to obtain the original k-mer

```
// Read node label
pos = csa[i];
label = sdsl::extract(csa, pos, pos+k−1);
reverse(label.begin(), label.end());
```

Code 2. Label extraction

Creating the bitvector L is very simple: we first initialize it with all ones then, considering we are in position i, insert a zero in position $i-1$ if the current label is equal to the previous one.

```
if (label == last_label) {
        L_bv[i−1] = 0;
}
```

Code 3. Construction of L

To flag the elements of W we simply have to check if the last $k-1$ characters of the current label are equal to those of the previous one. If this is the case then check if the edge points to the same node as a previous one and flag it accordingly. To keep track of which edges we already encountered we can keep a bitvector `chars_seen` of lenght $\sigma + 1$.

```
if (label.substr(1, k−1) ==
               last_label.substr(1, k−1)) {
    // Turn an edge into lowercase if it
    needs to be flagged
    if (chars_seen[to_int(edge)] == 1) {
        edge = tolower(edge);
    }
} else {
    sdsl::util::set_to_value(chars_seen, 0);
}
```

Code 4. Check for flags in W

Lastly, F can be temporarily stored using an integer vector while we copy it from `csa.C` and subtract k from every entry.

Finally, we can compress L, W, and F using the SDSL structure, in order to significantly reduce the memory occupied by the BOSS data structure. L, which was until now stored in a bitvector, can be stored with a RRR bit vector [4] which occupies $mH_0 + o(m)$ bits. W can be represented with a Wavelet Tree [5], which supports rank and select on strings with memory $mH_0 + 2\sigma \log m$. F, instead, can be simply compressed with the function `sdsl::util::bit_compress(F)`, which calculates $x = \max_i v[i]$ and then packs the entries of the original integer vector in l-bit integers, where $l = \lceil \log(x-1) \rceil + 1$.

Rank and select

All of the functions we are about to implement will be based on the use of rank and select queries:

- `rank(i, c)` returns the number of bits (for a bitvector) or characters (for a Wavelet trees) in the range $[0, i)$;
- `select(i, c)` returns the position of the i-th occurence of the symbol c.

We will use very often the two functions together. For example, two rank queries can count over a range, whereas two select queries can find a range. A rank and a select query can find a range where either a start point or end point are fixed.

Rank, select and standard array access can all be done in constant time when using the data structures discussed above.

Forward

`forward(i)`, together with `backward(i)`, is one of the two functions that will be needed to implement all the operations that our graph needs to support. It returns the index of the last edge of the node pointed to by edge i. It is useful to walk along the edges of the graph.

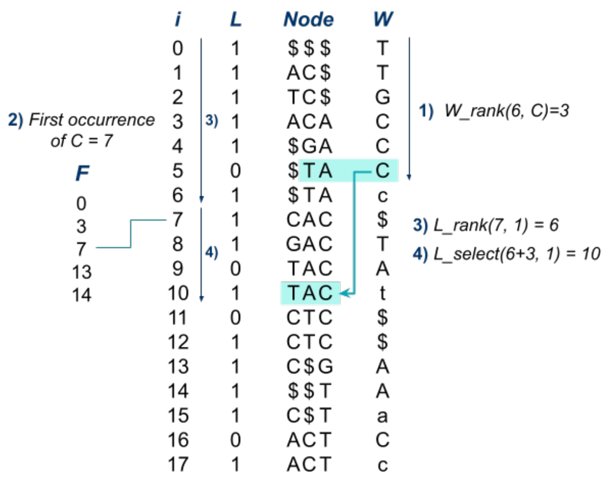


Figure 3. Steps for $forward(5) = 10$

Let's proceed with an example. Let's say we want to calculate `forward(10)`. Since the node labels have been sorted, they mantain the same relative order as the edge labels. Thus, walking along an edge simply consists in finding the corresponding relatively positioned node, e.g. since we want to follow the third edge labelled with `C`, we have to find the third node label ending with `C`.

Note though that the number of `C`s in the last column of the `Node` is slightly different from the number of `C`s in `W`, because of the fact that some edges point to the same node. Taking this into consideration, we only count non flagged edges in `W`, and also only work with edges that have $L[i] == 1$.

We can find the number of `C`s in W using rank, the first occurrence of `C` in the last column of the node labels from F, and then we can find the third node whose label ends with `C` using rank and select on L. The basic code is reported in Listing (5). The actual implementation only comprehends a few more line to take care of possible exceptions or invalid inputs.

All these operations take $O(1)$ time, so `forward` is calculated in constant time.

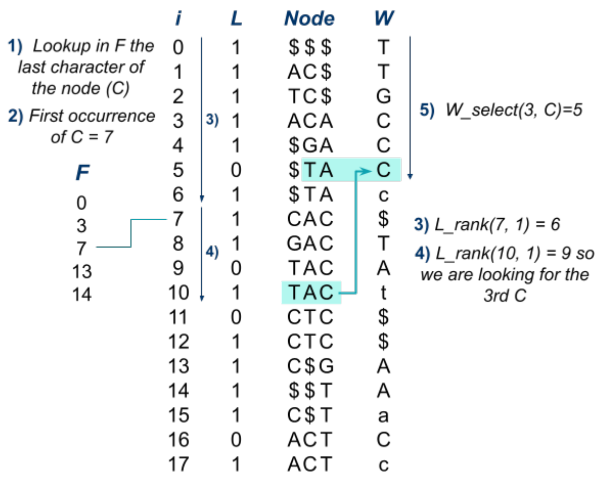
```
int forward(int i) {
    char edge = toupper(this->W[i]);
    int r = W_rank(i+1, edge);
    int fo = this->F[
      std::string("$ACGT").find(edge)
    ];

    return L_select(L_rank(fo, 1) + r, 1);
}
```

Code 5. Forward function implementation

Backward

Backward returns the index of the first edge that points to the node that edge i exits. It is very similar to forward, but calculated in the reverse order: we find the relative index of the node label first, and use that to find the corresponding edge (there may be more than one, but we consider only the first, the one which is not flagged).

The relative index of the node label is found through two rank queries on L. For example, if we want to calculate `backward(10)`, we can use the informations stored in F to find out that the last character of the node label is a `C` and that the first occurrence of `C` is in position 7. Then we count the number of ones in L to find out that $node[10]$ is the third one ending with `C`. The final step consists in using select on W to find the first incoming edge, i.e. the third one labelled with `C`.


1) Lookup in F the last character of the node (C)
2) First occurrence of C = 7
F
0
3
7
13
14
3)
4)
5) W_select(3, C)=5
3) L_rank(7, 1) = 6
4) L_rank(10, 1) = 9 so we are looking for the 3rd C

i	L	Node	W
0	1	$$$	T
1	1	AC$	T
2	1	TC$	G
3	1	ACA	C
4	1	$GA	C
5	0	$TA	C
6	1	$TA	c
7	1	CAC	$
8	1	GAC	T
9	0	TAC	A
10	1	TAC	t
11	0	CTC	$
12	1	CTC	$
13	1	C$G	A
14	1	$$T	A
15	1	C$T	a
16	0	ACT	C
17	1	ACT	c

Figure 4. Steps for $backward(10) = 5$

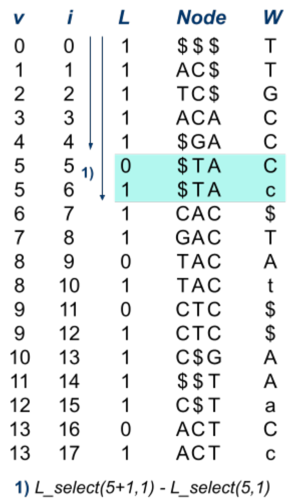
As in the case of the `forward` function, `backward` is computed in $\mathcal{O}(1)$ time as well.

```
int backward(int i) {
    char last_char;
    int fo, pointed;

    std::tie (last_char, fo) = F_lookup(i);
    int j = L_rank(i, 1) - L_rank(fo, 1) + 1;

    pointed = W_select(j, last_char);

    return pointed;
}
```

Code 6. Backward function implementation

Outdegree

This function accepts a node index $v \in [0, n-1]$ and returns the number of outgoing edges. Since all outgoing edges from the same node are contiguous we can take advantage of the fact that each distinct node is in a one-to-one correspondence to a 1 in L.



v	i	L	Node	W
0	0	1	$$$	T
1	1	1	AC$	T
2	2	1	TC$	G
3	3	1	ACA	C
4	4	1	$GA	C
5	5	0	$TA	C
5	6	1	$TA	c
6	7	1	CAC	$
7	8	1	GAC	T
8	9	0	TAC	A
8	10	1	TAC	t
9	11	0	CTC	$
9	12	1	CTC	$
10	13	1	C$G	A
11	14	1	$$T	A
12	15	1	C$T	a
13	16	0	ACT	C
13	17	1	ACT	c

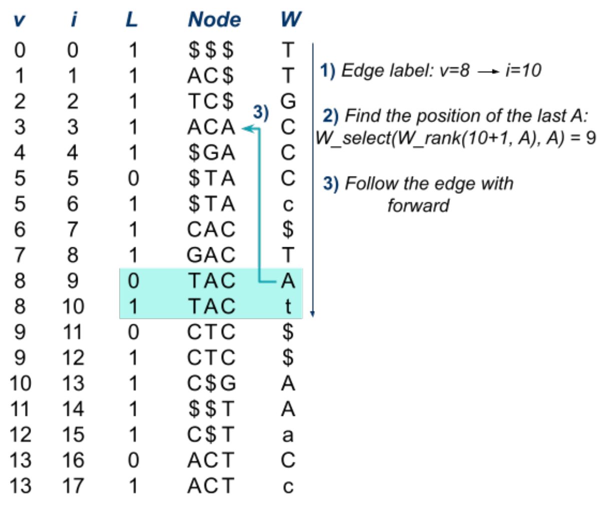
Figure 5. Steps for $outdegree(5) = 2$

In our example, since we want to calculate `outdegree(5)`, we use select to find the position of the sixth one in L, and subtract the position of the fifth one, giving `outdegree(5)=2`.

Outdegree is calculated in $\mathcal{O}(1)$ time as well, since we only use select queries.

Outgoing

`Outgoint(v, c)` returns the target node id walking along edge `c` starting from v, or -1 if this node doesn't exist. Let's see for example how to find `outgoing(8, A)`. First, we use select on L to find the corresponding edge id, then we use rank and select on W to find the position of the last A. We can then check if this edge falls into the range of edges outgoing from our selected node. If this is true, we can simply follow it using `forward`.



v	i	L	Node	W
0	0	1	$$$	T
1	1	1	AC$	T
2	2	1	TC$	G
3	3	1	ACA	C
4	4	1	$GA	C
5	5	0	$TA	C
5	6	1	$TA	c
6	7	1	CAC	$
7	8	1	GAC	T
8	9	0	TAC	A
8	10	1	TAC	t
9	11	0	CTC	$
9	12	1	CTC	$
10	13	1	C$G	A
11	14	1	$$T	A
12	15	1	C$T	a
13	16	0	ACT	C
13	17	1	ACT	c

1) Edge label: v=8 → i=10
2) Find the position of the last A: W_select(W_rank(10+1, A), A) = 9
3) Follow the edge with forward
3)

Figure 6. Steps for *outgoing*(8,'*A*') = 3

This process is reported in the following listing. Since we have flagged edges though, we also have to repeat the same exact operations with lowercase characters.

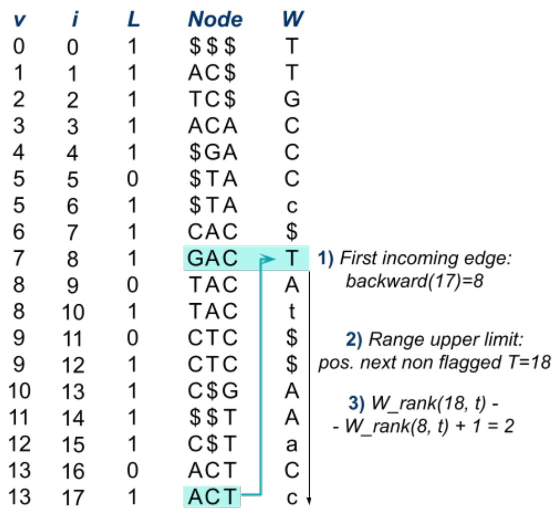
```
int outgoing(int v, char c) {
    int i = edge_id(v), out = -1;

    // Check for non flagged character
    int pos_last_char_upper = W_select(
        W_rank(i+1, toupper(c)),
        toupper(c)
    );

    if (pos_last_char_upper > edge_id(v-1) &&
     pos_last_char_upper <= i)
        out = node_id(
            forward(pos_last_char_upper)
        );

    return out;
}
```

Code 7. Outgoing function implementation

Outgoing is also calculated in $\mathcal{O}(1)$ time, since it involves a constant number of function calls.

Indegree

`Indegree(v)` returns the number of incoming edges to node v. We can easily find the first incoming (unflagged) edge by using `backward`. Then, we know that the next unflagged edge will point to a different node, so we simply have to count how many flagged edges there are in between.



v	i	L	Node	W
0	0	1	$$$	T
1	1	1	AC$	T
2	2	1	TC$	G
3	3	1	ACA	C
4	4	1	$GA	C
5	5	0	$TA	C
5	6	1	$TA	c
6	7	1	CAC	$
7	8	1	GAC	T
8	9	0	TAC	A
8	10	1	TAC	t
9	11	0	CTC	$
9	12	1	CTC	$
10	13	1	C$G	A
11	14	1	$$T	A
12	15	1	C$T	a
13	16	0	ACT	C
13	17	1	ACT	c

1) First incoming edge: backward(17)=8
2) Range upper limit: pos. next non flagged T=18
3) W_rank(18, t) - - W_rank(8, t) + 1 = 2

Figure 7. Steps for *indegree*(13) = 2

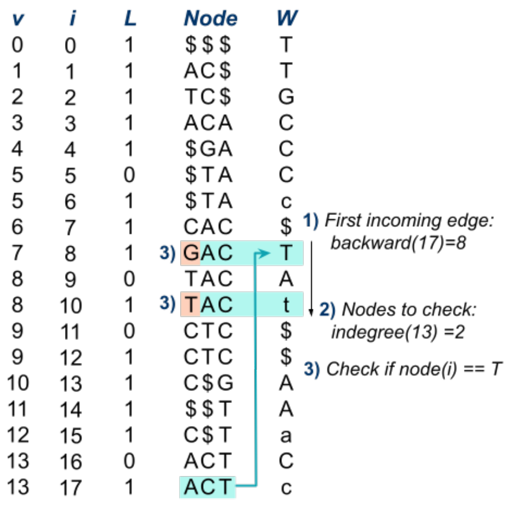
In the example in Figure (7) we are calculating `indegree(13)`. The first incoming edge labelled with `T` has $i = 8$, but there is no successive one. In this case we simply start counting the number of `T`s backwards from the end of W using rank. We have to remember to add 1 to account for the first unflagged edge.

```
int indegree(int v) {
    int first = backward(edge_id(v));

    char edge = this->W[first];
    int edges_prev = W_rank(first+1, edge);

    int next = W_select(edges_prev+1, edge);

    int indeg = W_rank(next, to_lower(edge))
   - W_rank(first, to_lower(edge)) + 1;


    return indeg;
}
```

Code 8. Indegree function implementation

Incoming

`Incoming(v,c)` returns the predecessor node that begins with the provided symbol. To implement it, we can use a similar approach to the one used for `indegree`. Infact, we already explained how to count the predecessors of a node. Now, instead of counting them with rank, we can use select to iterate over them, then label to check the first character.



v	i	L	Node	W
0	0	1	\$\$\$	T
1	1	1	AC\$	T
2	2	1	TC\$	G
3	3	1	ACA	C
4	4	1	\$GA	C
5	5	0	\$TA	C
5	6	1	\$TA	c
6	7	1	CAC	\$
7	8	1	GAC	T
8	9	0	TAC	A
8	10	1	TAC	t
9	11	0	CTC	\$
9	12	1	CTC	\$
10	13	1	C\$G	A
11	14	1	\$\$T	A
12	15	1	C\$T	a
13	16	0	ACT	C
13	17	1	ACT	c

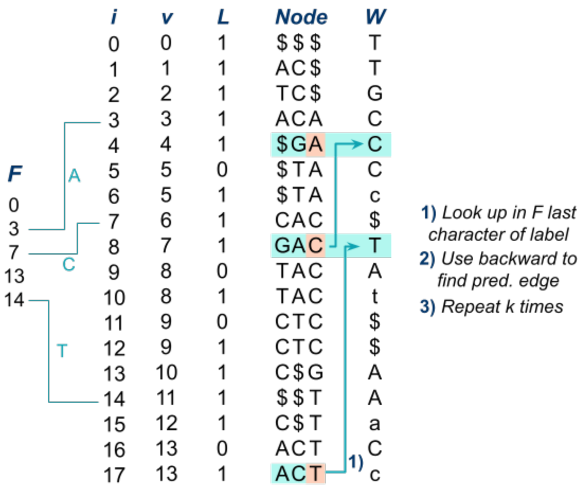
1) First incoming edge: backward(17)=8
2) Nodes to check: indegree(13) =2
3) Check if node(i) == T
3)
3)

Figure 8. Steps for *incoming*(13,'*T*') = 7

The most efficient way to implement this would be to use binary search instead of a linear scan, which would lead to $\mathcal{O}(k \log \sigma)$ time.

Label

The label function returns the label of node v. Remember that we are not storing the labels explicitely, so we'll have to take advantage of the information stored inside F. We can infact use it to find the last character of the node label in $\mathcal{O}(1)$ time.



i	v	L	Node	W
0	0	1	\$\$\$	T
1	1	1	AC\$	T
2	2	1	TC\$	G
3	3	1	ACA	C
4	4	1	\$GA	C
5	5	0	\$TA	C
6	5	1	\$TA	c
7	6	1	CAC	\$
8	7	1	GAC	T
9	8	0	TAC	A
10	8	1	TAC	t
11	9	0	CTC	\$
12	9	1	CTC	\$
13	10	1	C\$G	A
14	11	1	\$\$T	A
15	12	1	C\$T	a
16	13	0	ACT	C
17	13	1	ACT	c

F
0
3
7
13
14
A
C
T
1)
1) Look up in F last character of label
2) Use backward to find pred. edge
3) Repeat k times

Figure 9. Steps for *label*(13)

After this, we just use `backward` on the edge corresponding to our node to find an edge that pointed to this node, and repeat everything k times. The label is then found in $\mathcal{O}(k)$ time.

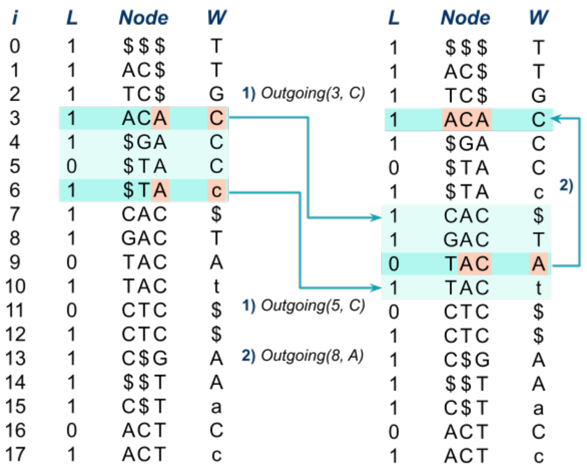
```
std::string label(int v) {
    std::string node_label;

    int i = edge_id(v), fo;
    char last_char;

    for (int j = 0; j < k; j++) {
        std::tie (last_char ,fo)=F_lookup(i);
        i = backward(i);
        node_label = last_char + node_label;
    }

    return node_label;
}
```

Code 9. Label function implementation

Index

The `index(s)` takes a k-mer s as input and returns the index v of the corresponding node if the k-mer is present in the input reads otherwise it returns -1.

To compute this, we start by considering the range of node labels ending with the first character of s. This is easy to find looking up in F. We then proceed by narrowing the search range through k successive applications of the `outgoing` operation.



i	L	Node	W	L	Node	W
0	1	\$\$\$	T	1	\$\$\$	T
1	1	AC\$	T	1	AC\$	T
2	1	TC\$	G	1	TC\$	G
3	1	ACA	C	1	ACA	C
4	1	\$GA	C	1	\$GA	C
5	0	\$TA	C	0	\$TA	C
6	1	\$TA	c	1	\$TA	c
7	1	CAC	\$	1	CAC	\$
8	1	GAC	T	1	GAC	T
9	0	TAC	A	0	TAC	A
10	1	TAC	t	1	TAC	t
11	0	CTC	\$	0	CTC	\$
12	1	CTC	\$	1	CTC	\$
13	1	C\$G	A	1	C\$G	A
14	1	\$\$T	A	1	\$\$T	A
15	1	C\$T	a	1	C\$T	a
16	0	ACT	C	0	ACT	C
17	1	ACT	c	1	ACT	c

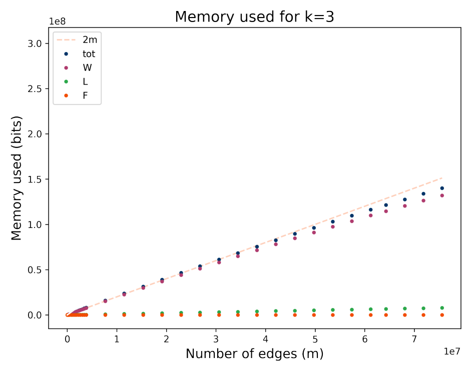
Figure 10. Steps for *index*(εACAε)

We can use select to find the first and last occurrence of the second character of s (flagged or unflagged) and then follow them using `outgoing`. If there are none of such occurrences we return -1, otherwise we continue narrowing the range following successive characters of s until, after k steps, we arrive at a single index v.

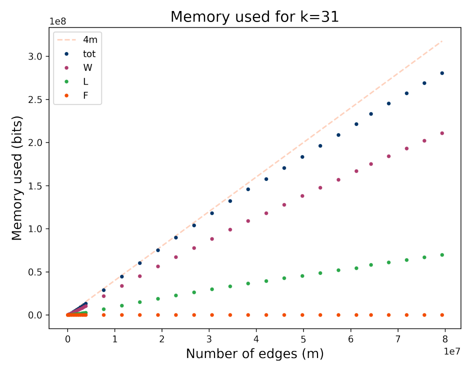
Since each step is done in constant time, in total `index` takes $O(k)$ time.

Results

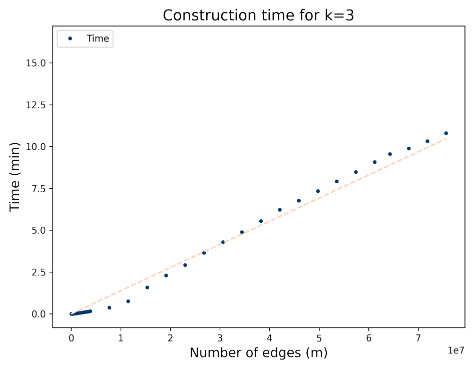
Figure (11) shows the memory occupied and time taken for construction of the BOSS data structure in the cases



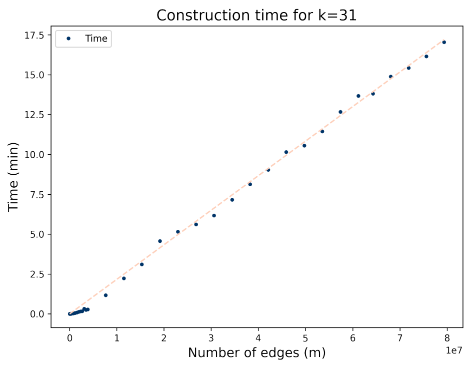
(a) Memory used for $k = 3$



(b) Memory used for $k = 31$



(c) Construction time for $k = 3$



(d) Construction time for $k = 31$

Figure 11. Memory used and construction time for varying k

$k = 3$ and $k = 31$. The memory occupied by each member of the data structure has been measured taking advantage of a function already implemented in the SDSL library, i.e. `size_in_bytes()`. The graphs in figure (11a) and (11b) show the total number of bits for increasing values of m. As we discussed during the construction of the class, the total number of bits should be $mH_0(L) + mH_0(W) + 2\sigma \log m + o(m)$. In the first case, for $k = 3$, the memory used for k-mer is under 2 bits, while in the second case is almost 4 bits, barely less than the uncompressed version, altough the difference starts to be more noticeable for larger values of m. The biggest difference lies in the memory occupied by L, due to the different values of $H_0(L)$ depending on k. Let's recall that

$$H_0(L) = \sum_{c \in \{0,1\}} \frac{n_c}{m} \log \frac{m}{n_c} = \frac{m-n}{m} \log \frac{m}{m-n} + \frac{n}{m} \log \frac{m}{n}$$

where n_c represents the number of occurrences of character c in L, and we indicated with n the number of ones, i.e. the number of distinct nodes.

We must remember that the maximum number of distinct nodes in a complete dBg is σ^k. So, in the case $k = 3$ the maximum number of ones in L is $n = 5^3 = 125$ (for DNA), while m reaches values of order 10^8 in our tests, and in real scenarios would be order of magnitudes bigger. This means that even if all 125 different k-mers are found in our reads, L will still contain a number of ones negligible with respect to the number of zeros, leading to a very small H_0.

The same conclusion cannot be drawn for $k = 31$. In this case we would have a maximum number of distinct nodes $n = 5^31 \approx 5 \cdot 10^{21}$, so the number of ones in L depends on the number of distinct k-mers in the reads, meaning that results will vary heavily depending on the input DNA string.

In the second row of figure (11) instead we can observe the time taken for the construction of the data structure. Let's recall that the major steps of the construction, the

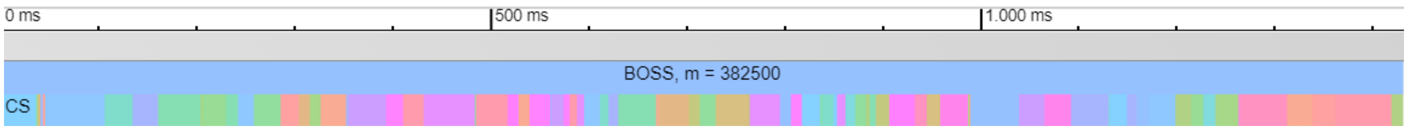


Figure 12. Breakdown of timing for $m = 382500$ and $k = 3$

more time consuming, are the creation of the CSA, the for loop, and the creation of the Wavelet Tree. In figure (12) we can see an example of a breakdown of the construction for $m = 382500$ and $k = 3$. The times are measured with a scope based timer which generates a JSON file to be read with Chrome tracing. The first blue block represents the CSA construction, then each different color represents a different k-mer, and finally we conclude with the wavelet tree in green.

As expected, since the time taken by all three stages increases linearly with m, the total construction time is also linear. Despite this, the multiplicative coefficient increases with k, because of the for loop, since the extraction of the label is not done in constant time, but it's proportional to k, i.e. to the number of characters to extract.

Conclusions

In conclusion, we have shown how to implement an efficent succinct representation of a de Bruijn graph also supporting all the needed navigation operations. The total teoretical space is $4m + o(m)$ for DNA but it can be compressed even further, depending on the value of k. The data structure can be constructed in $\mathcal{O}(m)$ time, but it shouldn't be difficult to reduce the construction time for example by parallelizing the for loop, which ends up being the most time consuming part of the algorithm.

References

[1] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," *Algorithms in Bioinformatics, Volume 7534 of Lecture Notes in Computer Science*, pp. 225–235, Sep. 2012. DOI: 10.1007/978-3-642-33122-0_18.

[2] A. Bowe. "Succinct de bruijn graphs." (Jul. 2013), [Online]. Available: https://www.alexbowe.com/succinct-debruijn-graphs/ (visited on 09/01/2024).

[3] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337.

[4] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Transactions on Algorithms*, vol. 3, no. 4, p. 43, Nov. 2007, ISSN: 1549-6333. DOI: 10.1145/1290672.1290680. [Online]. Available: http://dx.doi.org/10.1145/1290672.1290680.

[5] R. Grossi, A. Gupta, and J. Vitter, "High-order entropy-compressed text indexes," *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, Nov. 2002. DOI: 10.1145/644108.644250.

[6] P. Compeau, P. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, pp. 987–91, Nov. 2011. DOI: 10.1038/nbt.2023.

[7] P. Pevzner, H. Tang, and M. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 98, pp. 9748–53, Sep. 2001. DOI: 10.1073/pnas.171285098.

[8] R. Chikhi, J. Holub, and P. Medvedev, "Data structures to represent sets of k-long DNA sequences," *CoRR*, vol. abs/1903.12312, 2019. arXiv: 1903.12312. [Online]. Available: http://arxiv.org/abs/1903.12312.

[9] R. Chikhi, "A tale of optimizing the space taken by de bruijn graphs," in *Connecting with Computability*, Cham: Springer International Publishing, 2021, pp. 120–134.