Fragment Assembly in succint space

Compressing de Bruijn and Overlap graphs

Lapo Toloni

Seminar for the course Bioinformatics.
MsC in Computer Science, Università di Pisa, a.a. 2018/2019

2nd, August 2019

DNA Assembly

Within the last two decades, **assembling a genome from an enormous amount of reads** from various DNA sequencers has been one of the most challenging computational problems in molecular biology.

Such a problem is proved to be NP-hard and many algorithms have been devised to come to a solution both exact and approximated.

There are two types of algorithms that are commonly utilized by assemblers:

- **Greedy**, which aim for local optima,
- **Graph methods**, which aim for global optima.

Graph methods (Waterman and Gene Myers 1994)

Graph-method-assemblers come in two varieties: the ones based on **overlap graph** and the others based on **de Bruijn graphs**.

These methods represented an important step forward in sequence assembly, as they both use algorithms to reach a global optimum instead of a local one. Thanks to these theoretical advances computational genomics over large volumes of data became an easier problem to attack.

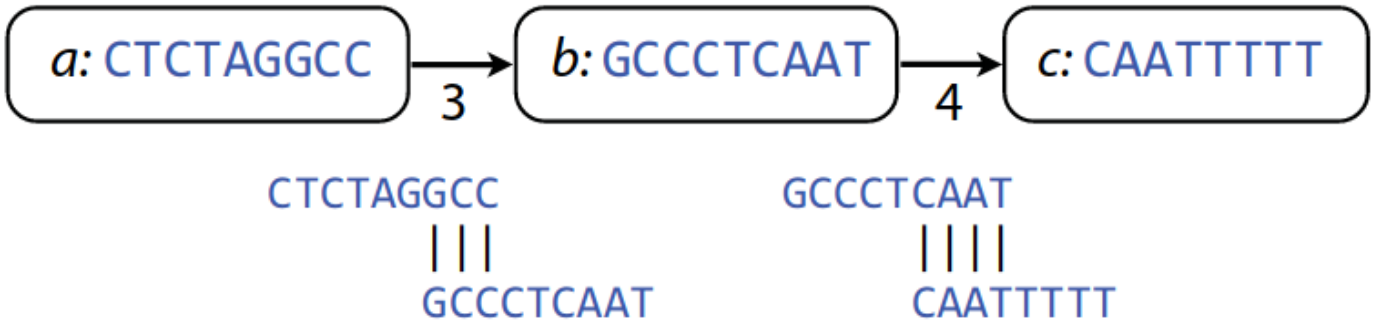
Both of these methods made progress towards better assemblies but nowadays the De Bruijn graph method has become the most popular due to the so called next-generation sequencing machines.

The Overlap graph

Most assembly algorithms from the *"Sanger era"* are based on the **overlap graph**.

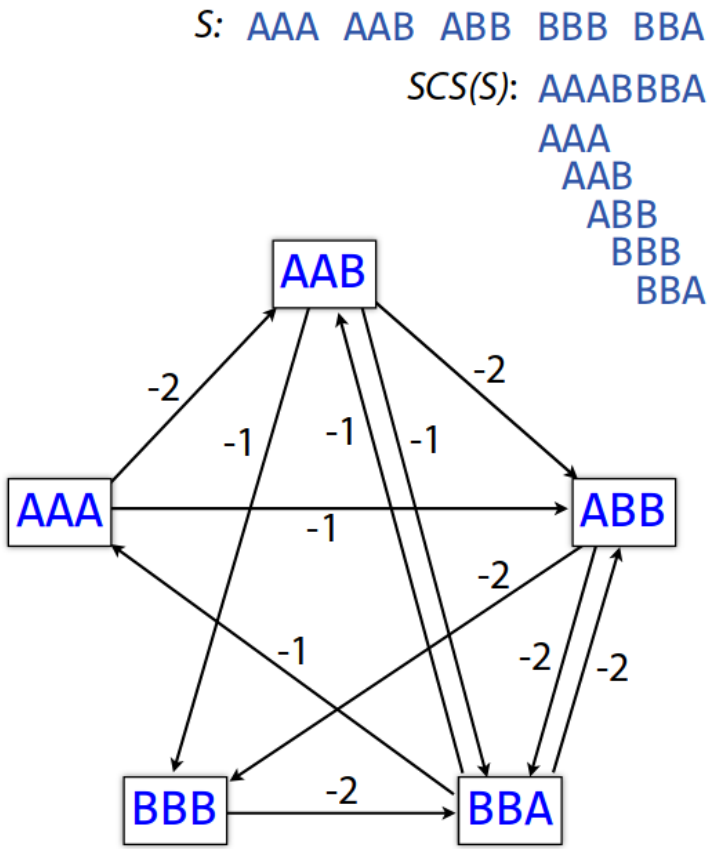
Such a graph has one node for each read produced by the sequencing process and two nodes are connected by a weighted edge iff the corresponding two reads have an overlap of enough length. *(this length is a parameter of the implementation)*

Read set $\mathcal{R} = \{CTCTAGGCC, GCCCTCAAT, CAATTTTT\}$



Shortest common superstring and TSP

By adding a minus sign in front of each edge label we get into a framework such that the assembly problem is reduced to the **Shortest common superstring problem** that indeed is equivalent to **TSP**, one of the most famous NP-HARD problems.



New Generation Sequencing

This strategy is too expensive to apply against the huge data from most recent **next-generation sequencers** (NGSs).

NGS machines can sequence vast amount of genome data.
It makes it computationally not tractable to compare all the pairs of reads to find the optimal solution for **SCS**.

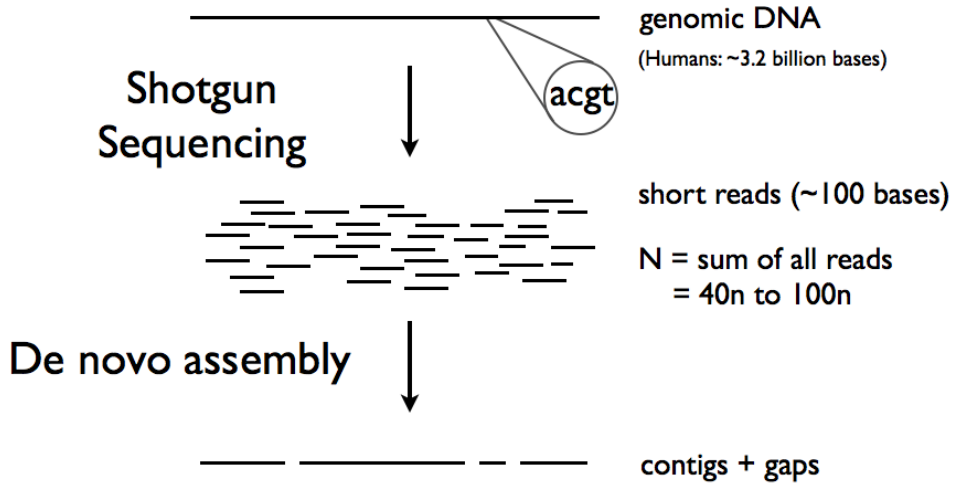
Moreover, most NGSs cannot read long DNA fragments (e.g., at most 200bp in the case of Illumina HiSeq2000), and their read lengths are not long enough to detect overlaps with enough lengths between reads.

New Generation Sequencing - Shotgun sequencing

One current popular NGS method is **shotgun sequencing**, which:

- clones the genome a bunch of times,
- then randomly breaks each clone into short segments

the fact that we randomly cut many clones of the same DNA segment should lead to have overlapping reads.



de Bruijn graph

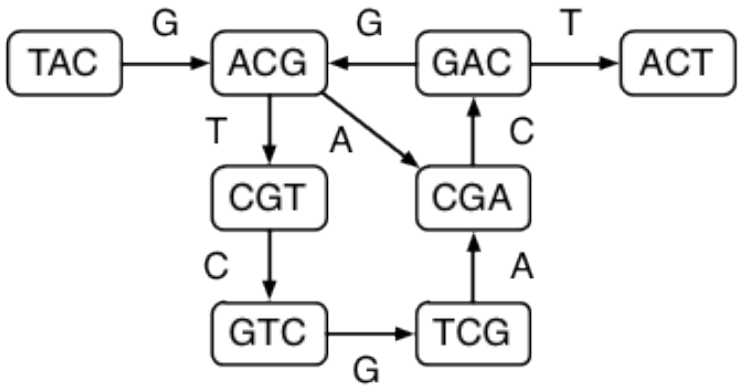
To conquer these problems, many recent assembler algorithms utilize a graph called the de Bruijn graph.

This kind of graphs is really suitable to represent a large network of overlapping **short read data**.

In this model each node represents a **k-mer** that exists in the reads, and an edge exists iff there is an exact overlap of length $\mathbf{k-1}$ between the corresponding k-mers.

Example read

$R = TACGACGTCGACT$



Eulerian path in de Bruijn graphs

After having built such a graph our original problem reduced to find **Eulerian paths** in it

Assuming **perfect sequencing** where each length-k substring is sequenced exactly once with no errors,
the naive "sliding-window" procedure to build the dBG always yields an Eulerian graph.

dBG PRO and CONS

The Single most important benefit of De Bruijn graph assemblers is speed and simplicity.

But since reads are immediately split into shorter k-mers

- Information about contexts longer than k is lost
- Read coherence is lost: some paths through De Bruijn graph are inconsistent with respect to input reads.
- dBGs need to be preprocessed because of an high probability of sequening errors (tips, bubbles ...)

Real world cases need something smaller and faster!

de Bruijn graphs can be constructed more efficiently than the overlap graph in many cases, but naive procedures still remain a bottleneck in term of used space.

This is because storing all the edges of the de Bruijn graph requires huge amount of memory.

Thus the focus of this presentation is on reducing the memory required for the de Bruijn graphs deploying some smart compression technique.

Succint data structures

A succint space data structure

- stores combinatorial objects using space asymptotical equal to the information theoretic lower bound.
- supports operations on its elements in $\mathcal{O}(1)$ time.

Note that if the input is taken from a set of $\mathcal{L}$ distinct combinatorial objects then its information-theoretic lower bound is $\lceil log\ \mathcal{L} \rceil$ bits.

Simple lower bounds

Example 1: **n elements sets** $\mathcal{S} \subseteq \{1, 2, .., n\}$:

$$log\ 2^n = n\ bits$$

Example 2: **length-n strings**:

$$n\ log\ \sigma\ bits\ (where\ \sigma\ is\ the\ alphabet\ size)$$

Example 3: **n nodes ordered trees**

$$log\ \frac{1}{2n-1}\binom{2n-1}{n-1} = 2n - \Theta(log\ n)$$

Some insights

For any data kind there exists a succint representation

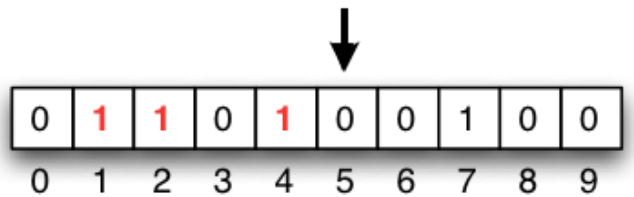
- simply obtained by enumerating and assign codes to every possible objects.
- that can be represented in $\lceil log\ \mathcal{L} \rceil$ bits

Unluckily not every representation is fine.
We need compressed structures upon we can implement fast operations.

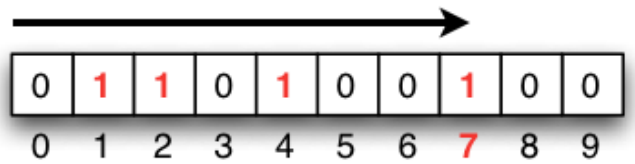
Rank / Select queries

Rank and select are the bread and butter of succinct data structures.
Infact almost every operation on these structures can be implemented combining rank and select queries.

- $\mathbf{rank_c(i)}$ = #occurences of **c** in the range [0,i]
- $\mathbf{select_c(i)}$ = position of the i-th occurrence of **c**.



Rank(5) = 3

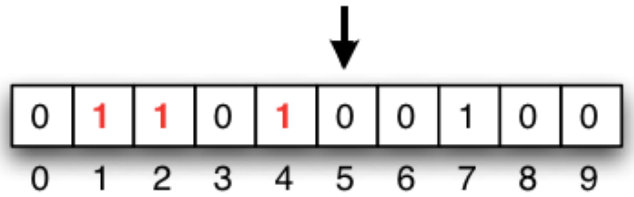


Select(4) = 7

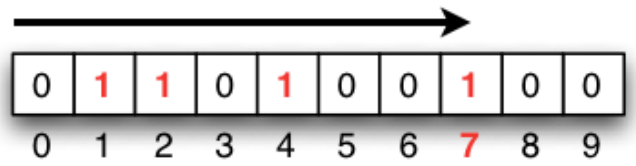
Rank / Select queries - Continued

Speaking of patterns of use, it may also help to keep in mind:

- rank is for **counting**;
  two rank queries can count over a range.
- select is for **searching**;
  two select queries can find a range



Rank(5) = 3



Select(4) = 7

Rank / Select queries - Performances

- Assuming that our alphabet size is $\sigma = polylog(N)$ ranks and selects can both be done in $\mathcal{O}(1)$ time by building over our bitvectors simple **RRR** structures. These additional structures requires just logarithmic space!
- for larger alphabets we could use wavelet trees achieving $\mathcal{O}(log\ \sigma)$. Even in this case by using huffman-shaped wavelet trees we can achieve sublinear space usage.

Succint de Bruijn Graphs - (Bowe, Sadakane, 2012)

Achievement: $4 + o(1)$ bits per edge (independent of k)

Inspiration taken from:

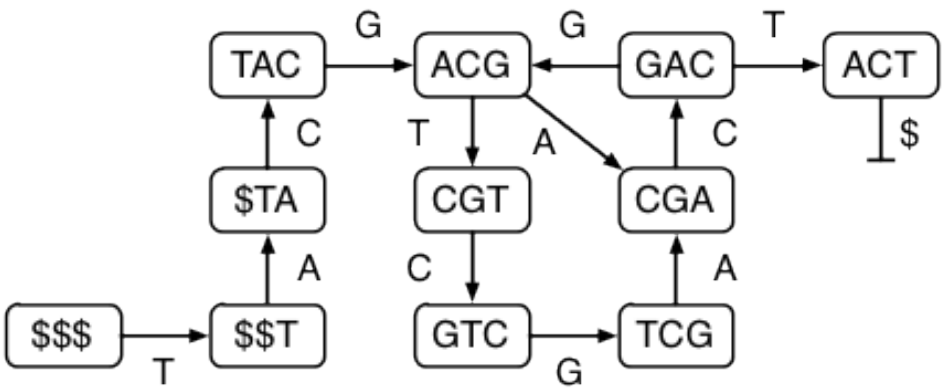
- BW Transform [Burrows, Wheeler 94]
- XBW Transform [Ferragina et al. 05]

Succint de Bruijn Graphs - Three arrays to represent a graph

Assuming that **m** is the number of edges of the graph

1. take each k-mer and **sort** them in reverse lexicographical order
   $\Rightarrow$ Array **Node** - Don't worry we won't store it explicitly.
2. store for each node the label of its outgoing edges
   $\Rightarrow$ Array **W**
3. store a bitvector **L** of size **m** whose entries are:
   $L[i] = 1 \Leftrightarrow Node[i] \neq Node[i+1]$
   i.e.: indicates the range in which Node refers the same node
   $\Rightarrow$ Array **L**

Succint de Bruijn Graphs - visualising BOSS



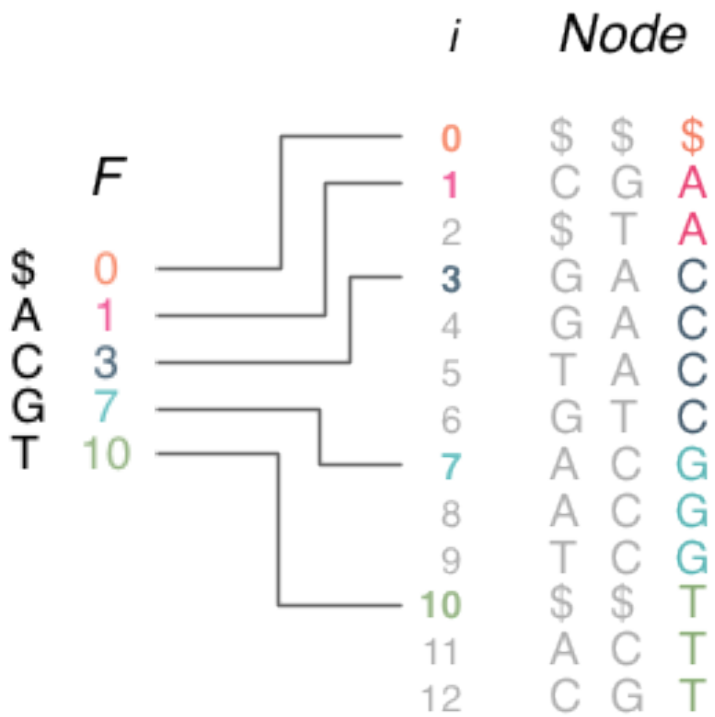
L	Node	W
1	$ $ $	T
1	C G A	C
1	$ T A	C
0	G A C	G
1	G A C	T
1	T A C	G-
1	G T C	G
0	A C G	A
1	A C G	T
1	T C G	A-
1	$ $ T	A
1	A C T	$
1	C G T	C

Optimizing space

Instead of storing the node labels we can save space by just storing the final column of the node labels.

Since the node labels are sorted, it is equivalent to store an array of first positions.

Array **Node** $\Rightarrow$ Array **F**



Total space used

Assuming that our graph has **m** edges we get:

1. Array **L** $\Rightarrow$ m bits
2. Array **W** $\Rightarrow$ $m \; log \; 2^{\sigma} \; = \; 3 \; m$ (in case of DNA alphabet)
3. Array **F** $\Rightarrow$ $\sigma \; log \; m \; = \; o(m)$

So total space required is $4 + o(1)$ bits per edge.

Fast travelling across the succint dBG

Recall that **DNA sequences can be reconstructed by moving between nodes of the dBG graph**.

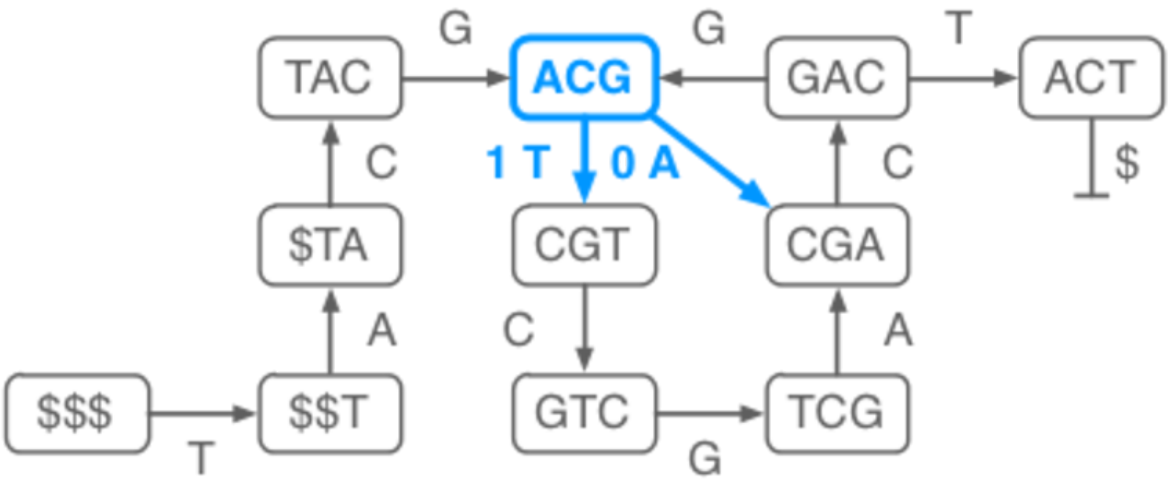
So we need some way, hopefully fast, to move through nodes and edges.

Thanks to two simple observations derived from how we succintly represented our graph and rank / select queries support for two of our three vectors we will achieve the wished speed of use.

BOSS properties - 1

Claim: (1)

All the edges going out from a node are next each other in the representation

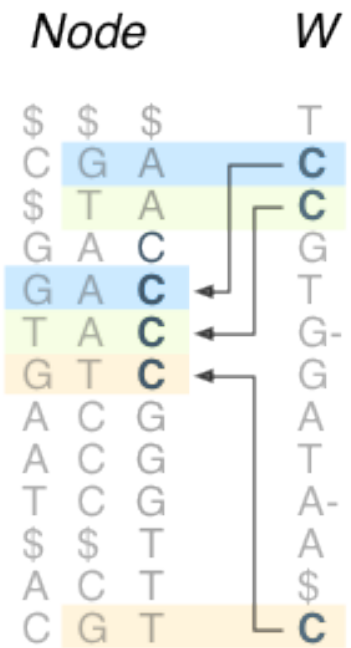


Node	W
$ $ $	T
C G A	C
$ T A	C
G A C	G
G A C	T
T A C	G-
G T C	G
A C G	A
A C G	T
T C G	A-
$ $ T	A
A C T	$
C G T	C

BOSS properties - 2

Claim: (2)

Nodes labels in the last column of **Node** maintain the same relative order as the edge labels in **W**.



Some other observations

1. In a dBG every node is defined by its previous k edges.
2. It may be the case that we need to disambiguate between edges that exit separate nodes but have the same label and enter the same node. (flagged labels in **W**)
3. Since a 1 in the **L** vector identifies a unique node, we can use this vector to index nodes (**HINT**: use $select_1$), whereas standard array accesses refer to edges.

Basic navigation: going forward and backward

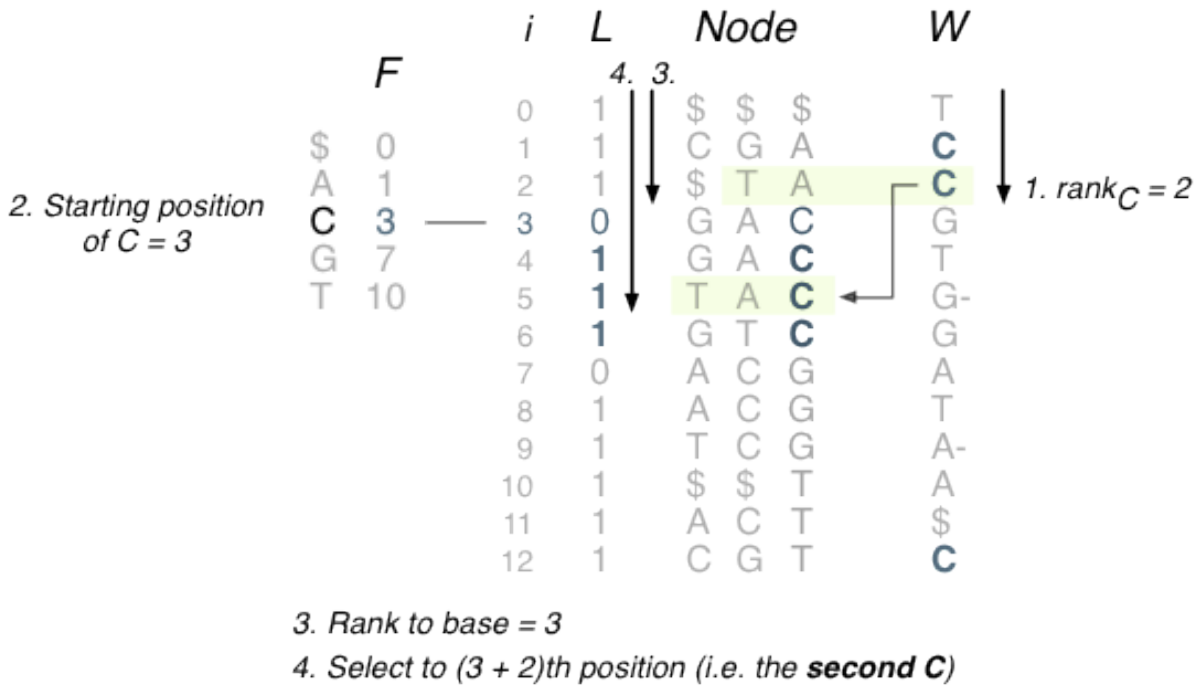
In order to support some public interface to navigate the graph, we first devise two simple procedures based on rank and select queries to work with edges:

- **forward(i)**: returns the index of the last edge of the node pointed by edge i.
- **backward(i)**: returns the index of the first edge that points to the node that the edge at i exits.

Forward

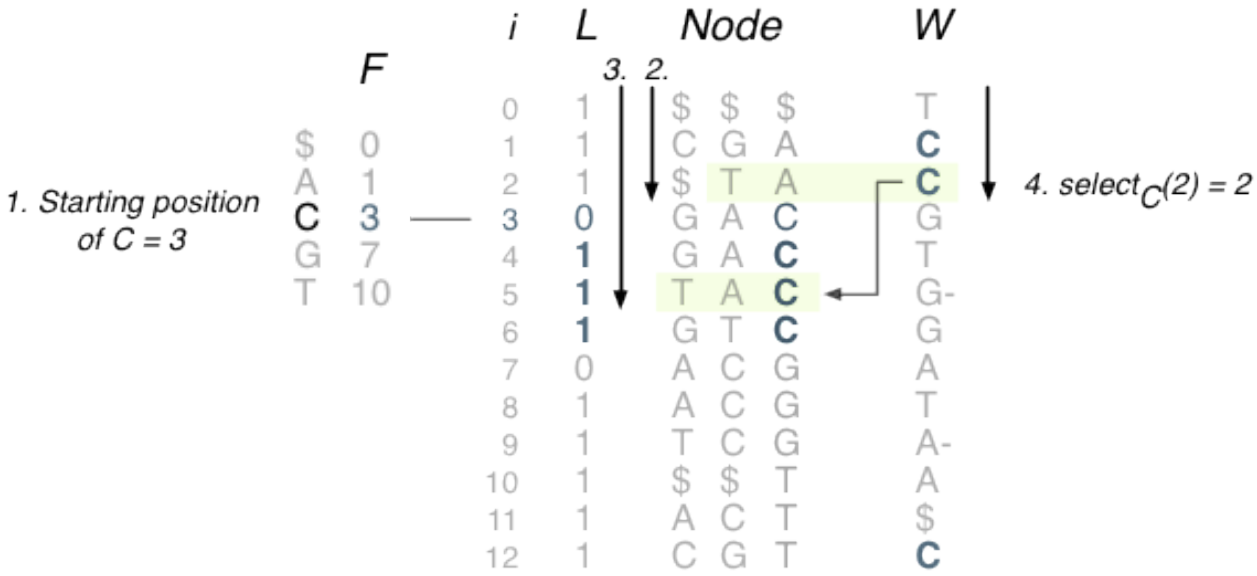
Thanks to **Claim 2** following an edge is simply finding the corresponding relatively positioned node.

Example: **forward(2)**



Backward

Again thanks to **Claim 2** following an edge backwardly is equivalent to the corresponding relatively positioned edge.
Example: **backward(5)**



Supported operations

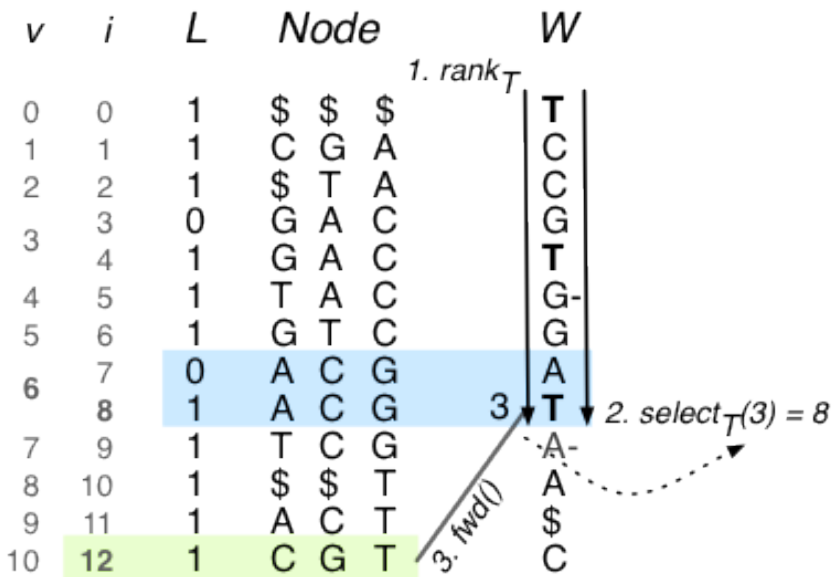
At this point we can use **forward(i)** and **backward(i)** to support the following operations:

- **outdegree(v)**: #outgoing edges for node v.
  $\mathcal{O}(1)$ time
- **outgoing(v, c)**: node reached from v through edge labelled c.
  $\mathcal{O}(1)$ time
- **indegree(v)**: #incoming edges for node v.
  $\mathcal{O}(1)$ time
- **incoming(v, c)**: predecessor node starting with symbol c, that has an edge to node v.
  $\mathcal{O}(\text{k } log\ \sigma)$ time
- **label(v)**: complete label of the node v.
  $\mathcal{O}(\text{k})$ time

Implementing outgoing(v, c)

Similar to **forward(i)** but now we want to move starting from a given node v.

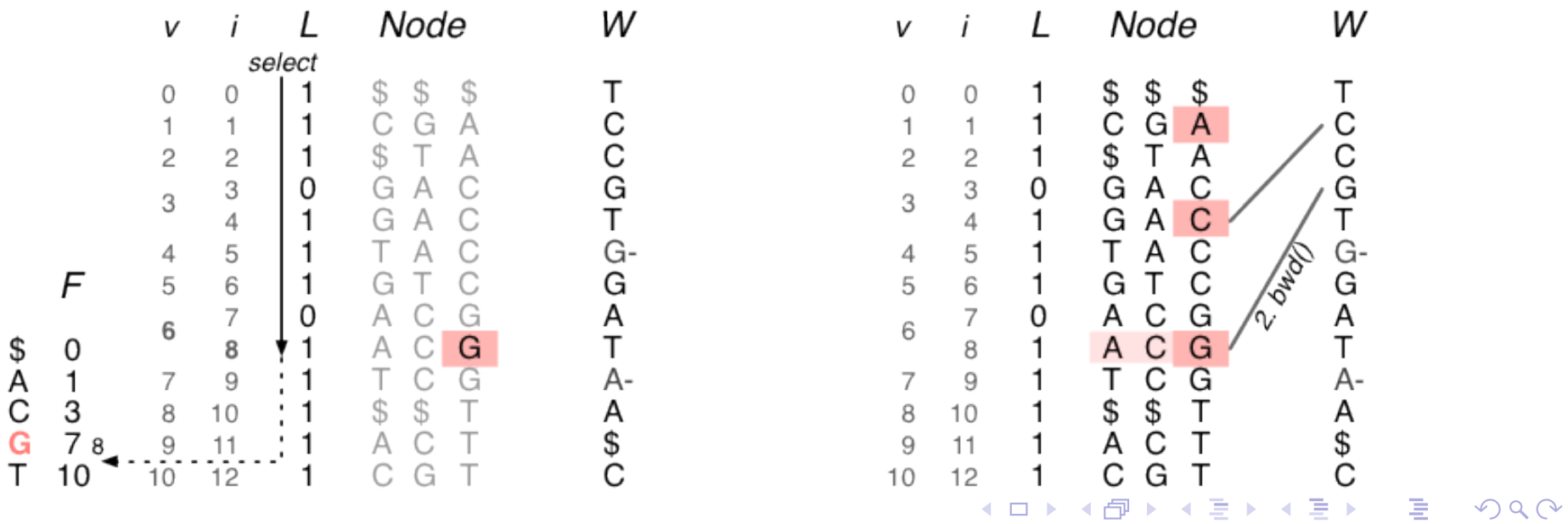
Example: **outgoing(6, T)**



Implementing label(v)

During a traversal we may want to print the node labels out.
We use the position of our node (**found using select(v)**) as a reverse lookup into **F** and then we do a series of k backward calls.

Example: **label(6)** note that in step 1 we do a select(6+1)!



Summarizing the results

- $m * (2 + log\ \sigma + o(1))$ bits, $4m\ +\ o(m)$ bits for DNA
- The size does not depend on k-mer length, so it is effective for large k too.
- useful navigation queries are perfomed in constant and in logarithmic time.

Where can we go from here?

We cannot know the best choice of k a priori

As seen before construction and navigation of the graph was a space and time bottleneck when acting naively.
But we saw how to solve it using compression techniques

Another problem emerges from the fact that state-of-the-art assemblers need to build the de Bruijn graph for multiple values of K to obtain better sequencing results

The need of the the right k

Lower k

- More connections
- Less chance of resolving small repeats
- Higher k-mer coverage

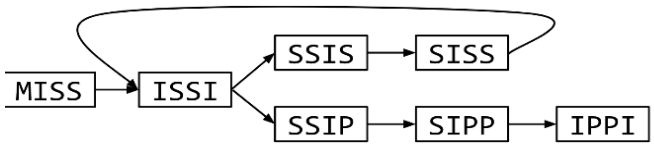
Higher k

- Less connections
- resolve small repeats
- Lower k-mer coverage

MISSIS SSISSI SSIPPI

All 4-mers:

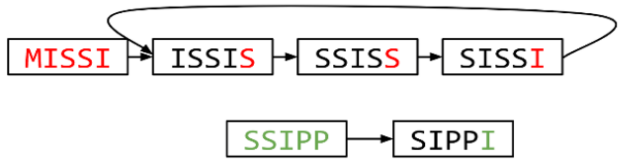
MISS ISSI SSIS SISS SSIP SIPP IPPI



MISSIS SSISSI SSIPPI

This time k = 5 k-mers:

MISSI ISSIS SSISS SISSI SSIPP SIPPI



MISSISSIS SSIPPI

Variable order dBG (Gagie et al., 2014)

In 2014 Gagie et al. showed how to augment a succinct de Bruijn graph representation letting us change order on the fly.

Just following a simple observation: deleting the first column of the BOSS-matrix the result is almost the BOSS matrix for a 2nd-order de Bruijn graph.

This truncated form of a higher order BOSS differs from the BOSS of a lower order in that some rows are repeated

This fact could prevent the BOSS representation from working properly.

Fixing it

Instead of trying to apply forward, backward and lastchar directly to nodes in the lower-order-graphs, we augment the BOSS representation of the original graph to support the following three queries:

- **shorter(v, k)** returns the node whose label is the last k characters of v's label;
- **longer(v, k)** lists nodes whose labels have length k $\leq$ K and end with v's label;
- **maxlen(v, a)** returns some node in the original graph whose label ends with v's. label, and that has an outgoing edge labelled a, or NULL otherwise.

If v is a node in the original graph then we can use the BOSS implementations of forward, backward and lastchar.

Otherwise, if v's label has length $k_v \leq k$ then:

- **fwd(v, a) = shorter(fwd(maxlen(v, a), a), k_v)**
- **bwd(v) = shorter(bwd(maxlen(longer(v,k_v+ 1), $*$)), k_v)**
- **lastchar(maxlen(v, $*$))**

Implementing shorter and longer

To implement shorter and longer, we store a wavelet tree over the sequence L^* in which $L^*[i]$ is the length of the longest common suffix of Node[i] and Node[i+1].

This takes $\mathcal{O}(log\ K)$ bits per (K + 1)-tuple in the matrix.
To save space, we can omit Ks in L^*, since they correspond to 0s in L and indicate that Node[i] and Node[i+1] are in the interval of the same node in the original graph;

the wavelet tree then takes $\mathcal{O}(log\ K)$ bits per node in the original graph and $\mathcal{O}(n\ log\ K)$ bits in total.

In our example, $L^* = [0, 1, 0, 3, 2, 1, 0, 3, 2, 0, 1, 1]$ (we can omit the 3s to save space).

Conclusions: Versatility of BOSS

The goodness of the BOSS representation was highlighted both by the original results and by all the extensions that came out in the following years.

Infact we just saw how to use BOSS to deal with variable order de Bruijn graphs.

But the improvements are not over ...

Concluding: simulating the overlap graph

In 2019 Dominguez et al [3] showed that BOSS can be extended even more arriving to simulate the overlap graph on the fly

The authors observed that overlaps between reads can be computed using voBOSS

Link between variable-order and overlaps

- extend a unary path using solid nodes as much as possible;

- if a solid node v without outgoing edges is reached, then decrease its order with shorter to retrieve the nodes that represent both a suffix of v and a prefix of some other read.

- from these nodes, retrieve the overlapping solid nodes of v by using forward and continue the graph right traversal from one of them.

When does shorter work correctly?

Since shorter does not ensure that the label of the output node appears as a prefix in the set of reads we need some precise conditions:

Lemma

In voBOSS, applying the operation shorter to a node v of order $k' \leq k$ will return a node u of order $k'' < k'$ that encodes a forward overlap for v

$\Leftrightarrow$

$u[1]$ is a linker node contained by v.

rBOSS fundamentals

We are ready to define the two operations needed to compute overlaps using voBOSS:

- **nextcontained(v)**: returns the greatest linker node v , in lexicographical order, whose llabel represents both a suffix of v and a prefix of some other node in G.
- **buildL(v,m)**: returns the set of all the linker nodes contained by v that represent a suffix of v of length $\geq m$.

the second operation is needed since a node might have more than one contained linker node, and those linkers whose **llabel** is of length $\geq m$ represent edges in the overlap graph.

Final remarks

- if we chose $k = z + 1$ to build voBOSS we are simulating the full overlap graph in compressed space.
  The edges are not stored explicitly, but computed on the fly by first obtaining L = buildL(v, m), and then following the dBG outgoing edges of every l $\in$ L.
- This extension substitutes the **LCS** over **L** with a compacted trie to achieve fast implementation of **buildL(v,m)**.

References

1. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI), pages 225–235, 2012.
2. Christina Boucher, Alexander Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In Proc. 25th Data Compression Conference (DCC), pages 383–392, 2015.
3. Diego Díaz-Domínguez, Travis Gagie, Gonzalo Navarro. Simulating the DNA String Graph in Succinct Space

Thanks for the attention!

Any question?