Space-efficient construction of succinct de Bruijn graphs

Felipe A. Louza

University of São Paulo, Brazil

Joint work with
Lavinia Egidi and Giovanni Manzini.

LSD/LAW
London, 6-7 Feb. 2019

Outline

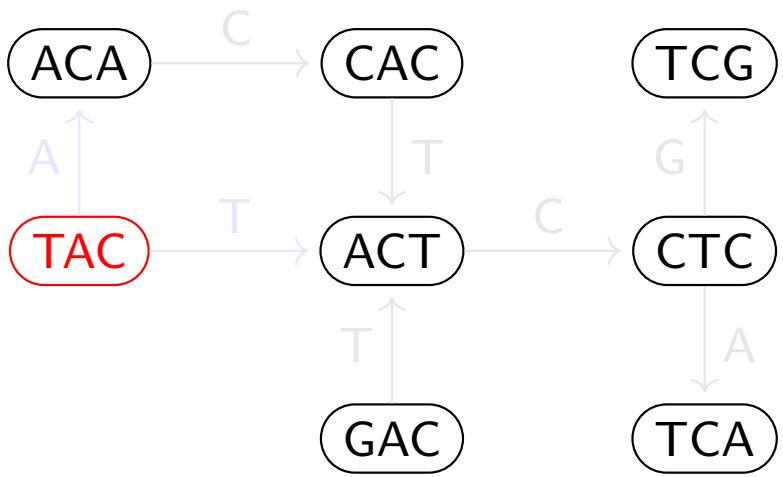
de Bruijn graphs (dBGs)

Definitions:

- Given a collection of strings $\mathcal{S}$, a *de Bruijn* graph of order k is a directed graph containing:
  - a node v for every **unique** k-**mer** $v[1]...v[k]$ in $\mathcal{S}$.
  - an edge (u, v) with label $v[k]$ if there is a $(k+1)$-**mer** $u[1]...u[k]v[k]$ in $\mathcal{S}$.

Example:

- $\mathcal{S} = \{$TACACT, TACTCA, GACTCG$\}$



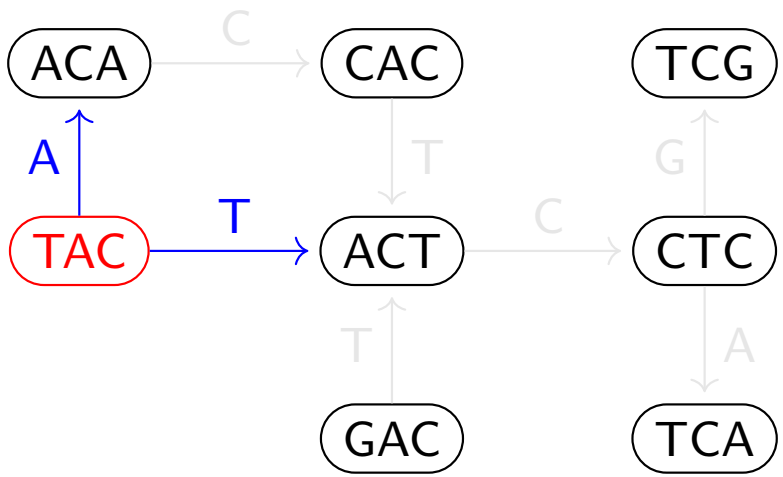
de Bruijn graphs (dBGs)

Definitions:

- Given a collection of strings $\mathcal{S}$, a *de Bruijn* graph of order k is a directed graph containing:
  - a node v for every **unique** k-**mer** $v[1]...v[k]$ in $\mathcal{S}$.
  - an edge (u, v) with label $v[k]$ if there is a $(k+1)$-**mer** $u[1]...u[k]v[k]$ in $\mathcal{S}$.

Example:

- $\mathcal{S} = \{$TACACT, TACTCA, GACTCG$\}$



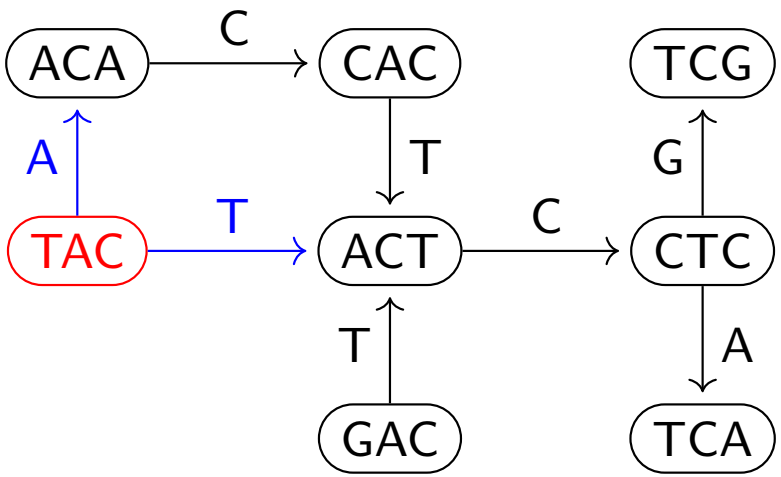
de Bruijn graphs (dBGs)

Definitions:

- Given a collection of strings $\mathcal{S}$, a *de Bruijn* graph of order k is a directed graph containing:
  - a node v for every **unique** k-**mer** $v[1]...v[k]$ in $\mathcal{S}$.
  - an edge (u, v) with label $v[k]$ if there is a $(k+1)$-**mer** $u[1]...u[k]v[k]$ in $\mathcal{S}$.

Example:

- $\mathcal{S} = \{$TACACT, TACTCA, GACTCG$\}$



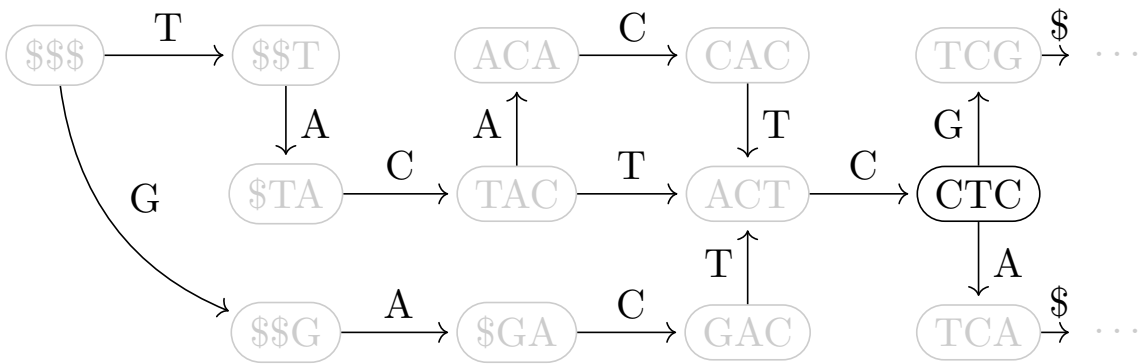
Succinct representation of dBGs:

BOSS*:

- In [Bowe *et al.*, WABI 2012] introduced a succinct representation for dBGs in space $\mathcal{O}(|E| \log \sigma)$ bits.
- BOSS representation:
  - Outgoing edges of each v_i are encoded into the substring $W_i = x_1[i]\ x_2[i]\ \ldots\ x_\sigma[i]$
  - W_i are concatenated considering the order of the reversed labels: $W = \ldots W_{\$\$\$}\, W_{\$\$T}\, W_{\$\$G}\, W_{\$TA}\, W_{ACA}\, W_{\$GA}\, W_{TAC}\, W_{CAC}\, W_{GAC}\, W_{ACT}\, W_{TCG}\, W_{CTC}\, W_{TCA} \ldots$

Example:

- $\mathcal{S} = \{$ TACACT, TACTCA, GACTCG$\}$



*for the authors' initials

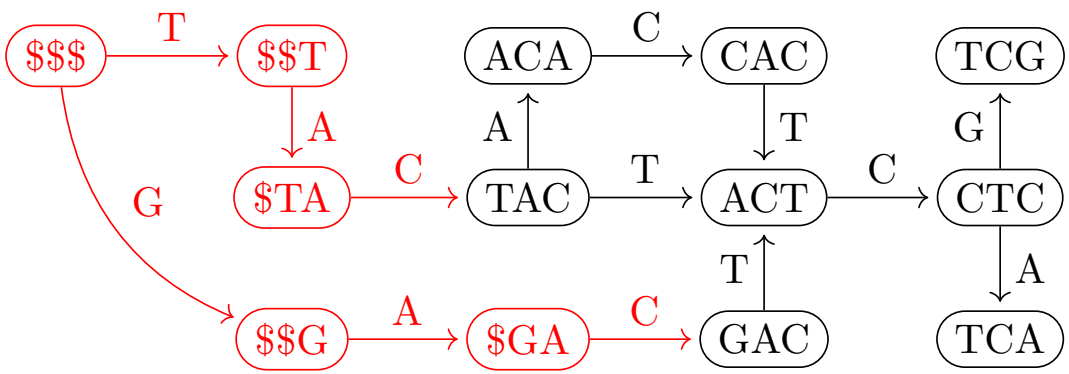
Succinct representation of dBGs:

BOSS*:

- In [Bowe *et al.*, WABI 2012] introduced a succinct representation for dBGs in space $\mathcal{O}(|E| \log \sigma)$ bits.
- BOSS representation:
  - Outgoing edges of each v_i: are encoded into the substring $W_i = v_i[k] \cdots v_k[k]$
    $W_i = \mathsf{AT}, W_j = \mathsf{AG}$
  - W_i are concatenated considering the order of the reversed labels $\overleftarrow{v_1}, \ldots, \overleftarrow{v_n}$
    $\mathsf{TACGACT}$

Example:

- $\mathcal{S} = \{\$\$\$\mathsf{TACACT}, \$\$\$\mathsf{TACTCA}, \$\$\$\mathsf{GACTCG}\}$



For convenience, we add k copies of a symbol \$ at the beginning of each string s_i.

Succinct representation of dBGs:

BOSS*:

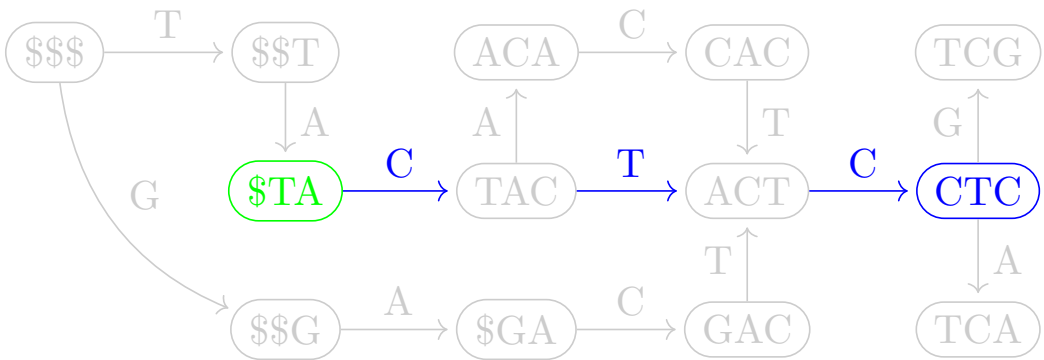
- In [Bowe *et al.*, WABI 2012] introduced a succinct representation for dBGs in space $\mathcal{O}(|E| \log \sigma)$ bits.
- BOSS representation:
  - Outgoing edges of each v_i: are encoded into the substring $W_i = v_i[k] \cdot \cdot \cdot v_i[k]$

    $W_i = \mathsf{AT}, W_j = \mathsf{AG}$
  - W_i are concatenated considering the order of the reversed labels $\overleftarrow{v_i} = v_i[k] \cdot \cdot \cdot v_i[1]$

    $\overleftarrow{TAC} \prec \overleftarrow{CTC}$

Example:

- $\mathcal{S} = \{\$\$\$\mathsf{TACACT}, \$\$\$\mathsf{TACTCA}, \$\$\$\mathsf{GACTCG}\}$



The label of every node can be recovered.

Succinct representation of dBGs:

BOSS*:

- In [Bowe *et al.*, WABI 2012] introduced a succinct representation for dBGs in space $\mathcal{O}(|E| \log \sigma)$ bits.
- BOSS representation:
  - Outgoing edges of each v_i: are encoded into the substring $W_i = v_j[k] \dots v_k[k]$

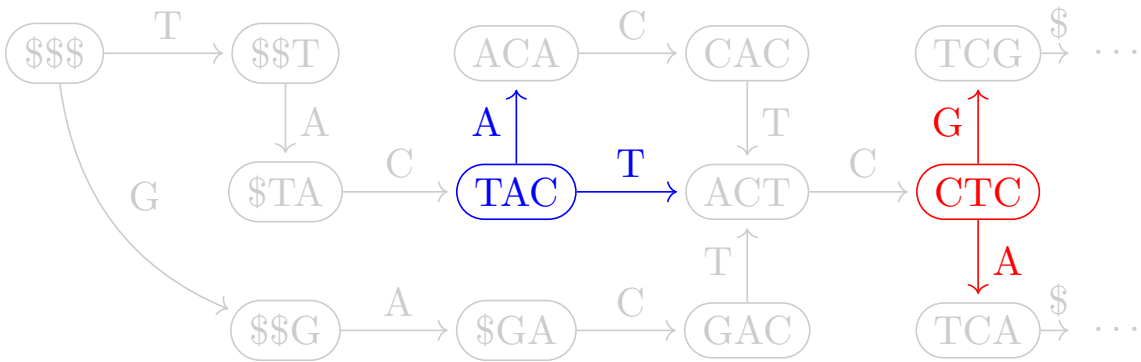
$$W_i = \mathsf{AT}, W_j = \mathsf{AG}$$

  - W_i are **concatenated** considering the order of the **reversed labels** $\overleftarrow{v_i} = v_i[k] \dots v_i[1]$

$$\overleftarrow{TAC} \prec \overleftarrow{CTC}$$

Example:

- $\mathcal{S} = \{\$\$\$\mathsf{TACACT}, \$\$\$\mathsf{TACTCA}, \$\$\$\mathsf{GACTCG}\}$



Succinct representation of dBGs:

BOSS*:

- In [Bowe *et al.*, WABI 2012] introduced a succinct representation for dBGs in space $\mathcal{O}(|E| \log \sigma)$ bits.
- BOSS representation:
  - Outgoing edges of each v_i: are encoded into the substring $W_i = v_j[k] \dots v_k[k]$

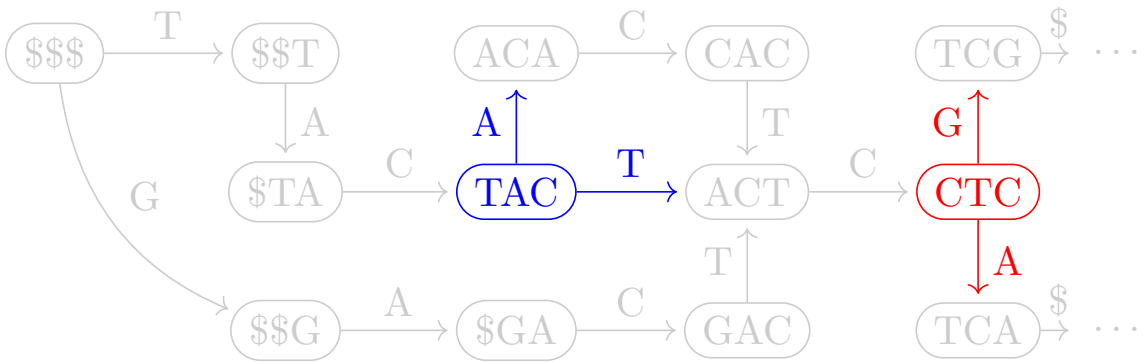
$$W_i = \mathsf{AT}, W_j = \mathsf{AG}$$

  - W_i are **concatenated** considering the order of the **reversed labels** $\overleftarrow{v_i} = v_i[k] \dots v_i[1]$

$$\overleftarrow{TAC} \prec \overleftarrow{CTC}$$

Example:

- $\mathcal{S} = \{\$\$\$\mathsf{TACACT}, \$\$\$\mathsf{TACTCA}, \$\$\$\mathsf{GACTCG}\}$

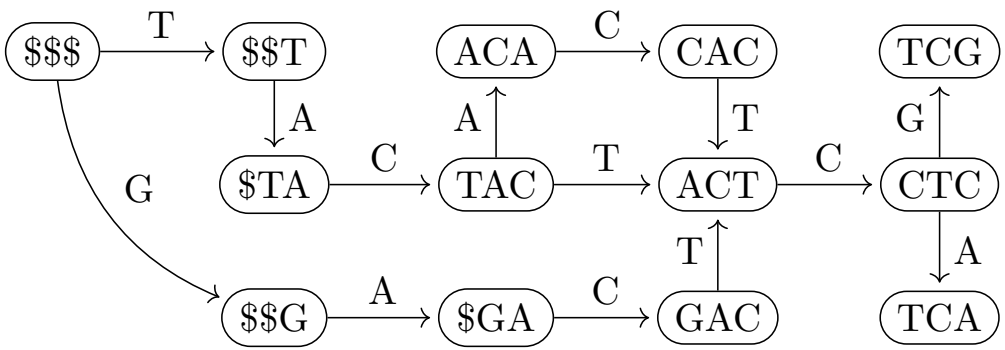


Succinct representation of dBGs:

BOSS:

- Nodes $v_i = \underline{v_i[1]...v_i[k]}$ are sorted by their **reversed labels** $\overleftarrow{v_i} = \underline{v_i[k]...v_i[1]}$
- We mark the position of the last outgoing edge of each node.
- We mark as negative ($-$) incoming edges with the same label (except the first).

last	Nodes	W
0	\$ \$ \$	G
1	\$ \$ \$	T
1	A C A	C
1	T C A	\$
1	\$ G A	C
1	\$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	\$ \$ G	A
1	T C G	\$
1	\$ \$ T	A
1	A C T	C

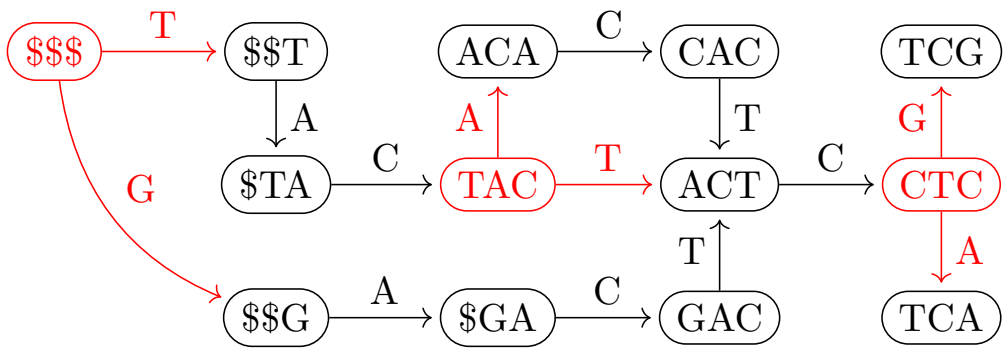


Succinct representation of dBGs:

BOSS:

- Nodes $v_i = \underline{v_i[1]...v_i[k]}$ are sorted by their **reversed labels** $\overleftarrow{v_i} = \underline{v_i[k]...v_i[1]}$
- We mark the position of the last outgoing edge of each node.
- We mark as negative ($-$) incoming edges with the same label (except the first).

last	Nodes	W
0	\$ \$ \$	G
1	\$ \$ \$	T
1	A C A	C
1	T C A	\$
1	\$ G A	C
1	\$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	\$ \$ G	A
1	T C G	\$
1	\$ \$ T	A
1	A C T	C

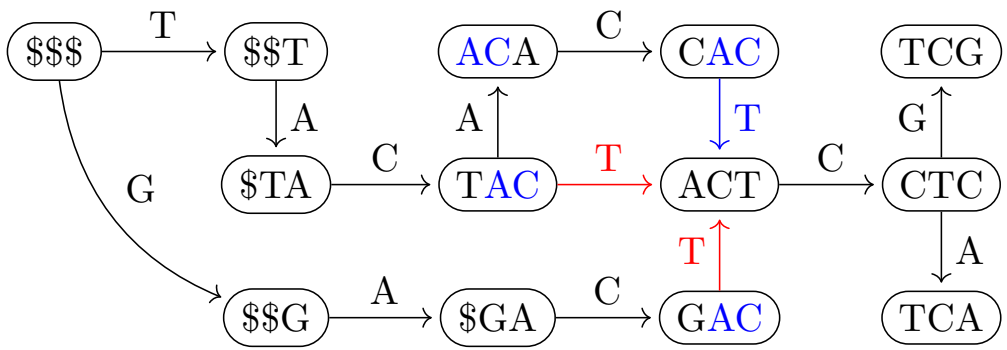


Succinct representation of dBGs:

BOSS:

- Nodes $v_i = v_i[1]...v_i[k]$ are sorted by their **reversed labels** $\overleftarrow{v_i} = v_i[k]...v_i[1]$
- We mark the position of the last outgoing edge of each node.
- We mark as negative ($-$) incoming edges with the same label (except the first).

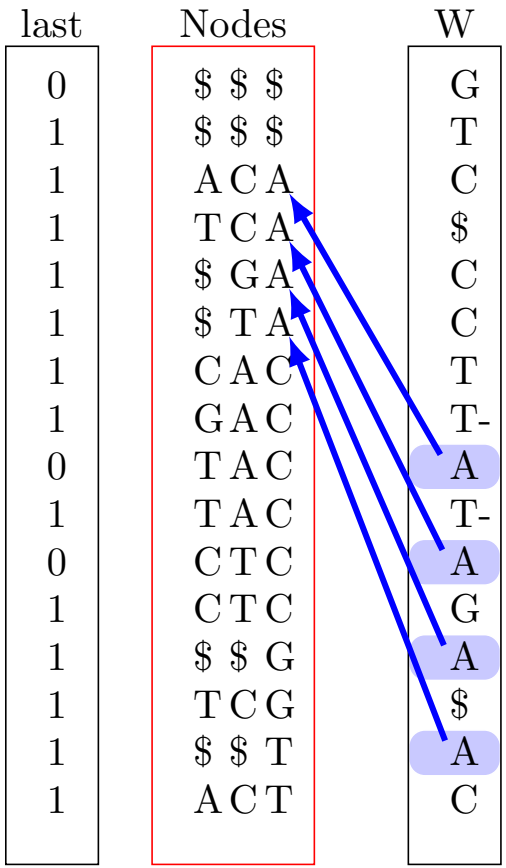
last	Nodes	W
0	\$ \$ \$	G
1	\$ \$ \$	T
1	A C A	C
1	T C A	\$
1	\$ G A	C
1	\$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	\$ \$ G	A
1	T C G	\$
1	\$ \$ T	A
1	A C T	C

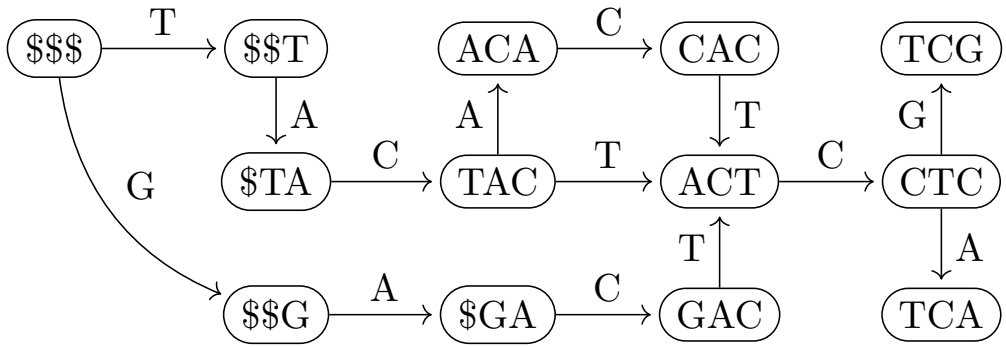


Succinct representation of dBGs:

BOSS:

- LF-mapping between the positive symbols in W and the $Nodes[k]$ (with last $= 1$).
- Fast navigation operations: Outdegree, Outgoing, Indegree and Incoming.
- Small space: $\mathcal{O}(m \log \sigma) + m + o(m)$ bits for rank and select operations.



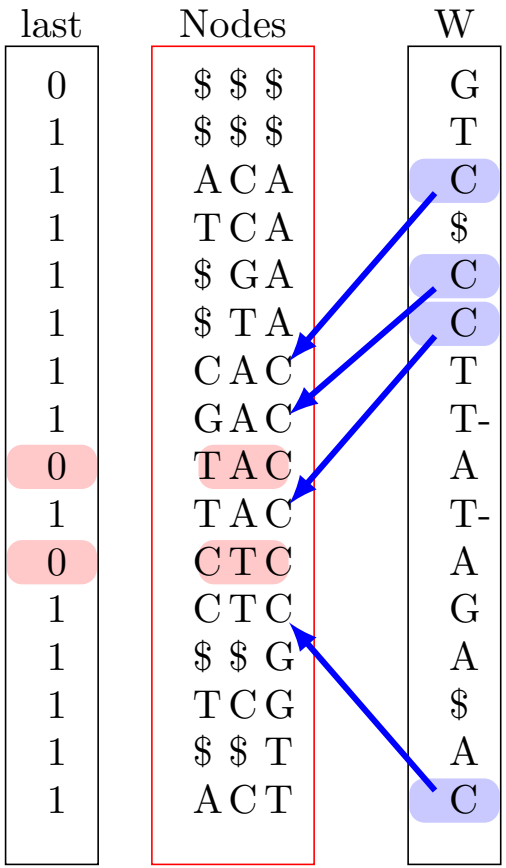


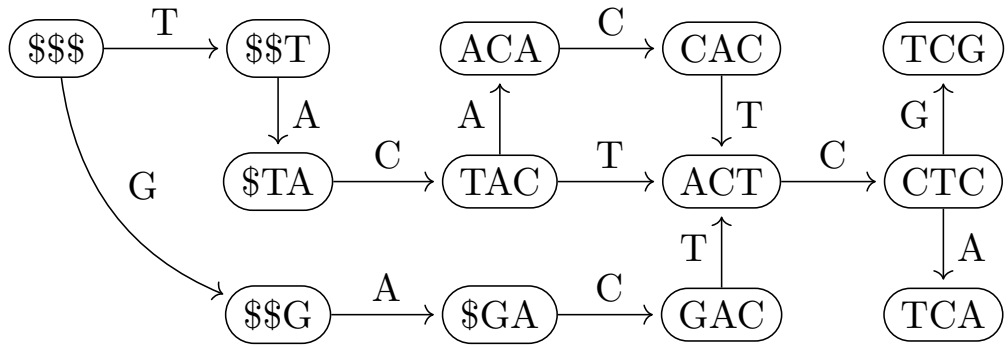
Similar to the BWT and XBW.

Succinct representation of dBGs:

BOSS:

- LF-mapping between the positive symbols in W and the $Nodes[k]$ (with last $= 1$).
- Fast navigation operations: Outdegree, Outgoing, Indegree and Incoming.
- Small space: $\mathcal{O}(m \log \sigma) + m + o(m)$ bits for rank and select operations.



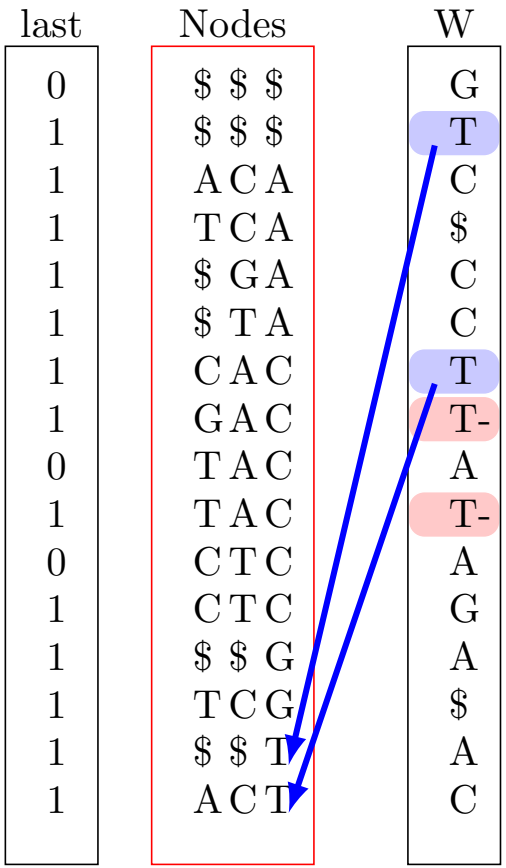


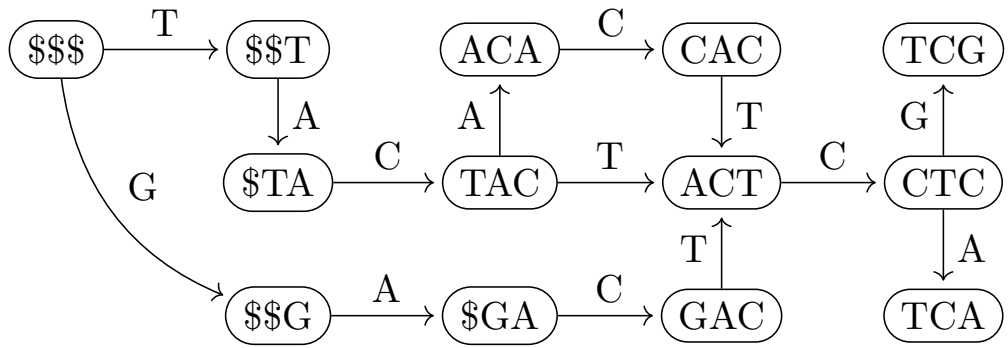
Select operation.

Succinct representation of dBGs:

BOSS:

- LF-mapping between the positive symbols in W and the $Nodes[k]$ (with last $= 1$).
- Fast navigation operations: Outdegree, Outgoing, Indegree and Incoming.
- Small space: $\mathcal{O}(m \log \sigma) + m + o(m)$ bits for rank and select operations.



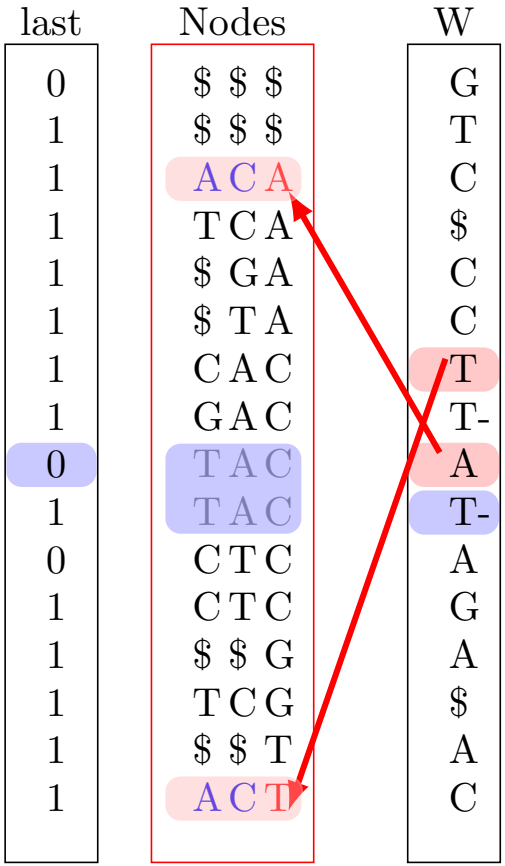


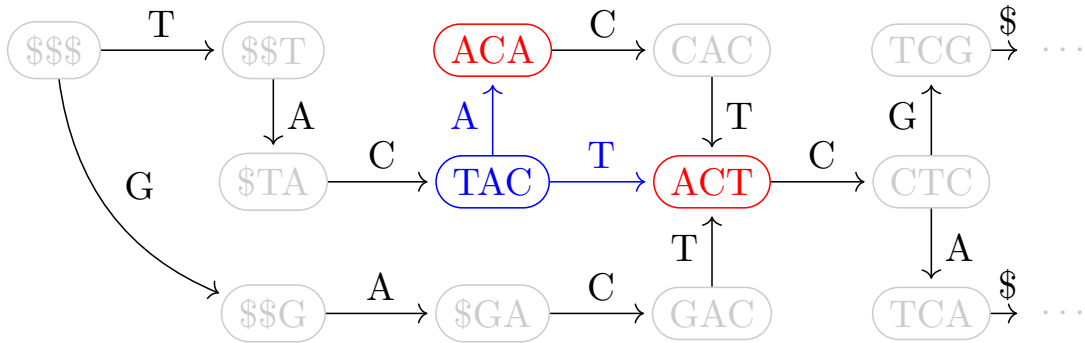
Select operation.

Succinct representation of dBGs:

BOSS:

- LF-mapping between the positive symbols in W and the $Nodes[k]$ (with last $= 1$).
- Fast navigation operations: Outdegree, Outgoing, Indegree and Incoming.
- Small space: $\mathcal{O}(m \log \sigma) + m + o(m)$ bits for rank and select operations.



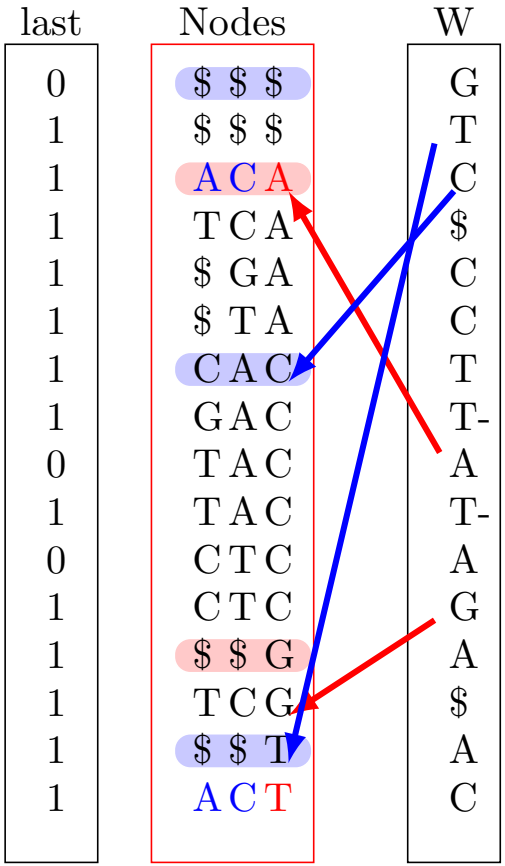


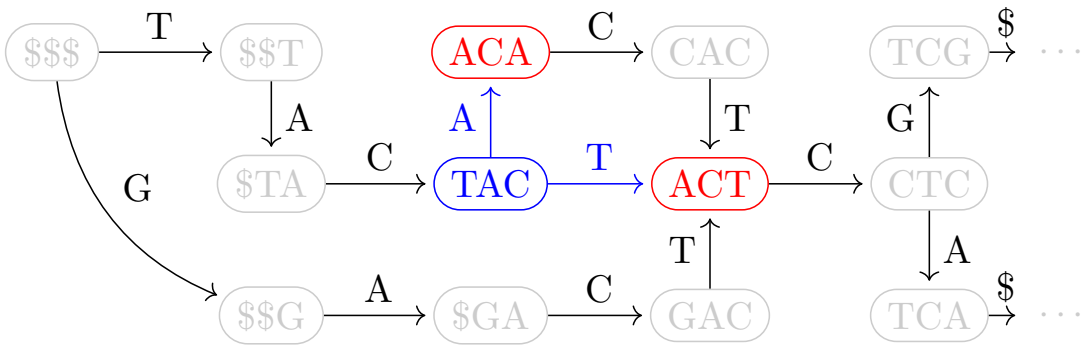
Outgoing edges.

Succinct representation of dBGs:

BOSS:

- LF-mapping between the positive symbols in W and the $Nodes[k]$ (with last $= 1$).
- Fast navigation operations: Outdegree, Outgoing, Indegree and Incoming.
- Small space: $\mathcal{O}(m \log \sigma) + m + o(m)$ bits for rank and select operations.





We don't need matrix Nodes, we can use counters $C[1, \sigma]$.

Outline

BOSS construction

Radix sort:

- We can sort all k-mers of collection $\mathcal{S}$, with total length N.
  - Radix sorting (from 2nd symbol of $(k+1)$-mers): in $O(N \cdot k)$ time.

$h=1$	$h=2$	$h=3$	last	Nodes	W
G $ $ $	G $ $ $	G $ $ $	0	$ $ $	G
T $ $ $	T $ $ $	T $ $ $	1	$ $ $	T
C A G $	C A C A	C A C A	1	A C A	C
C A T $	$ A C T	$ A C T	1	T C A	$
C A C A	C A G $	C A G $	1	$ G A	C
$ A C T	C A T $	C A T $	1	$ T A	C
T C A C	T C A C	T C A C	1	C A C	T
A C T C	T C A G	T C A G	1	G A C	T-
G C T C	A C A T	A C A T	0	T A C	A
T C A G	T C A T	T C A T	1	T A C	T-
A C A T	A C T C	A C T C	0	C T C	A
T C A T	G C T C	G C T C	1	C T C	G
$ G C T	$ G C T	$ G C T	1	$ $ G	A
A G $ $	A G $ $	A G $ $	1	T C G	$
A T $ $	A T $ $	A T $ $	1	$ $ T	A
C T C A	C T C A	C T C A	1	A C T	C

BOSS construction

BWT and LCP array: [Egidi et al., WABI 2018]

- Given the BWT and LCP array for the reversed strings:
  - We can compute $\langle W, \text{last} \rangle$ in $O(N)$ time: sequential scan over BWT+LCP.
  - We need only the k-truncated BWT and LCP array.

BWT	sorted suffixes	LCP
G	$ $ $	2
T	$ $ $	3
T	$ $ $	3
C	A C A T $	0
$	A C T C A T $	2
C	A G $	1
C	A T $	1
C	A T $	5
T	C A C A T $	0
T	C A G $	2
T	C A T $	2
A	C A T $	6
G	C T C A G $	1
A	C T C A T $	4
A	G $	0
$	G C T C A G $	1
A	T $ $	0
A	T $ $	4
$	T C A C A T $	1
C	T C A G $	3
C	T C A T $	3

last	Nodes	W
0	$ $ $	G
1	$ $ $	T
1	A C A	C
1	T C A	$
1	$ G A	C
1	$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	$ $ G	A
1	T C G	$
1	$ $ T	A
1	A C T	C

BOSS construction

BWT and LCP array: [Egidi et al., WABI 2018]

- Given the BWT and LCP array for the reversed strings:
  - We can compute $\langle W, \text{last} \rangle$ in $O(N)$ time: sequential scan over BWT+LCP.
  - We need only the k-truncated BWT and LCP array.

BWT	sorted suffixes	LCP
G	$ $ $	2
T	$ $ $	3
T	$ $ $	3
C	A C A T $	0
$	A C T C A T $	2
C	A G $	1
C	A T $	1
C	A T $	5
T	C A C A T $	0
T	C A G $	2
T	C A T $	2
A	C A T $	6
G	C T C A G $	1
A	C T C A T $	4
A	G $	0
$	G C T C A G $	1
A	T $ $	0
A	T $ $	4
$	T C A C A T $	1
C	T C A G $	3
C	T C A T $	3

last	Nodes	W
0	$ $ $	G
1	$ $ $	T
1	A C A	C
1	T C A	$
1	$ G A	C
1	$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	$ $ G	A
1	T C G	$
1	$ $ T	A
1	A C T	C

BOSS construction

BWT and LCP array: [Egidi et al., WABI 2018]

- Given the BWT and LCP array for the reversed strings:
  - We can compute $\langle W, \text{last} \rangle$ in $O(N)$ time: sequential scan over BWT+LCP.
  - We need only the k-truncated BWT and LCP array.

BWT	sorted suffixes	LCP
G	$ $ $	2
T	$ $ $	3
T	$ $ $	3
C	A C A T $	0
$	A C T C A T $	2
C	A G $	1
C	A T $	1
C	A T $	5
T	C A C A T $	0
T	C A G $	2
T	C A T $	2
A	C A T $	6
G	C T C A G $	1
A	C T C A T $	4
A	G $	0
$	G C T C A G $	1
A	T $ $	0
A	T $ $	4
$	T C A C A T $	1
C	T C A G $	3
C	T C A T $	3

last	Nodes	W
0	$ $ $	G
1	$ $ $	T
1	A C A	C
1	T C A	$
1	$ G A	C
1	$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	$ $ G	A
1	T C G	$
1	$ $ T	A
1	A C T	C

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time:
- Peak memory:

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 22×, 4× and 3× slower.
- Peak memory:

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 22×, 4× and 3× slower.
- Peak memory: 1.3× larger, 54× and 110× smaller.

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 15×, 2.5× and 2× slower.
- Peak memory: 1.7× larger, 13.6× and 28× smaller.

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	
512MB	BWT+BOSS	981.49		991.03		991.71		← se-gap
	COSMO	**65.69**	303 MB	**396.76**	6.8 GB	**479.01**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 7×, 1.4× and 1.2× slower.
- Peak memory: 3.5× larger, 3.4× and 7× smaller.

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	
512MB	BWT+BOSS	981.49		991.03		991.71		← se-gap
	COSMO	**65.69**	303 MB	**396.76**	6.8 GB	**479.01**	14.4 GB	
2GB	BWT+BOSS	560.95		569.61		564.69		← gap
	COSMO	**78.65**	578 MB	**403.96**	6.8 GB	**479.39**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 22×,
- Peak memory: 1.7× larger,

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	
512MB	BWT+BOSS	981.49		991.03		991.71		← se-gap
	COSMO	**65.69**	303 MB	**396.76**	6.8 GB	**479.01**	14.4 GB	
	BWT+BOSS*	363.94						← k-gap
2GB	BWT+BOSS	560.95		569.61		564.69		← gap
	COSMO	**78.65**	578 MB	**403.96**	6.8 GB	**479.39**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 22×, 4×
- Peak memory: 1.7× larger, 13.6×

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	
512MB	BWT+BOSS	981.49		991.03		991.71		← se-gap
	COSMO	**65.69**	303 MB	**396.76**	6.8 GB	**479.01**	14.4 GB	
	BWT+BOSS*	363.94		709.25				← k-gap
2GB	BWT+BOSS	560.95		569.61		564.69		← gap
	COSMO	**78.65**	578 MB	**403.96**	6.8 GB	**479.39**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

BOSS construction

Experiments:[1]

1. Counting k-mers (DSK) and Radix sorting (COSMO) $\leftarrow \mathcal{O}(N \cdot k)$ time.
2. BWT and LCP construction (egap) and BOSS construction $\leftarrow \mathcal{O}(N)$ time.

Results:

- Running time: 22×, 4× and 3× slower.
- Peak memory: 1.7× larger, 13.6× and 28× smaller.

RAM parameter		k=10		k=30		k=50		
		time (sec)	memory	time (sec)	memory	time (sec)	memory	
128MB	BWT+BOSS	1,486.03		1,501.62		1,496.59		← em-gap
	COSMO	**67.15**	99 MB	**397.53**	6.8 GB	**465.93**	14.4 GB	
512MB	BWT+BOSS	981.49		991.03		991.71		← se-gap
	COSMO	**65.69**	303 MB	**396.76**	6.8 GB	**479.01**	14.4 GB	
	BWT+BOSS*	363.94		709.25		1,025.96		← k-gap
2GB	BWT+BOSS	560.95		569.61		564.69		← gap
	COSMO	**78.65**	578 MB	**403.96**	6.8 GB	**479.39**	14.4 GB	

[1] DNA sequences of length 100 (500MB).

Outline

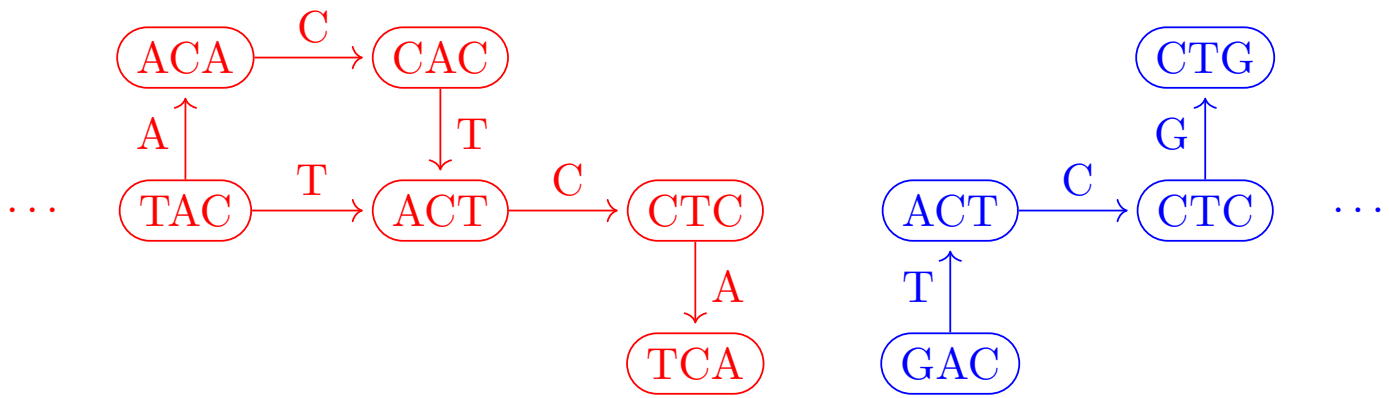
Merging dBGs

Merging BOSS representations

- Suppose we are given the BOSS representation of two* de Brujin graphs $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ from the collections of strings $\mathcal{C}_0$ and $\mathcal{C}_1$
- We compute the BOSS for $\mathcal{C}_{01} = \mathcal{C}_0 \cup \mathcal{C}_1$ directly, that is, without decoding G_0 and G_1 and encode G_{01}.

Example:

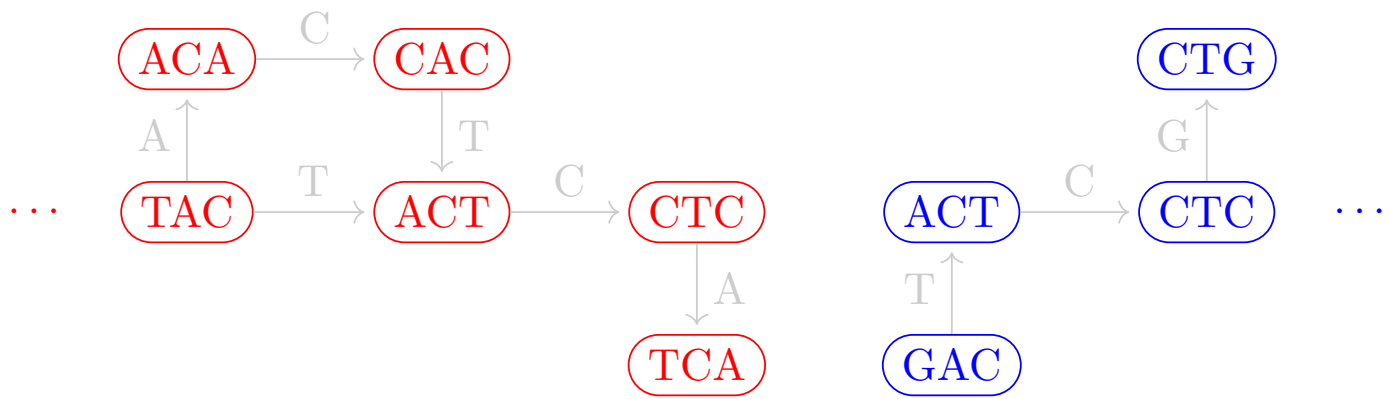
- $\mathcal{S}_1 = \{\$\$\$\text{TACACT}, \$\$\$\text{TACTCA}\} \cup \{\$\$\$\text{GACTCG}\}$



Merging dBGs

Merging BOSS representations

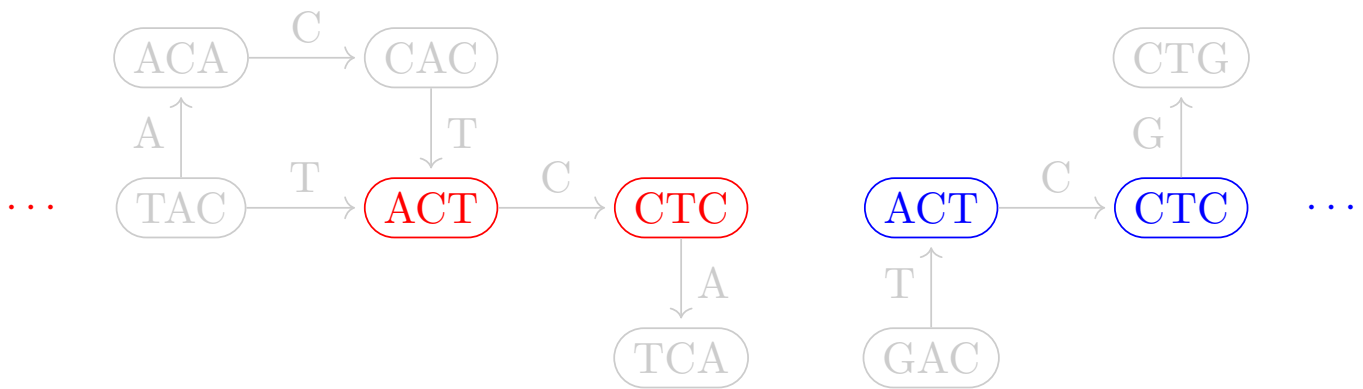
- Tasks:
  1. Merge the nodes in G_0 and G_1 according the order of their k-mers,
     $$\overleftarrow{v_1} \prec \cdots \prec \overleftarrow{v_{n_0}} \qquad \text{and} \qquad \overleftarrow{w_1} \prec \cdots \prec \overleftarrow{w_{n_1}}$$
  2. Recognize when two nodes in G_0 and G_1 refer to the same k-mer, and
     $$\overleftarrow{v_i} \;?=\; \overleftarrow{w_j}$$
  3. Properly merge and update W and last.



Merging dBGs

Merging BOSS representations

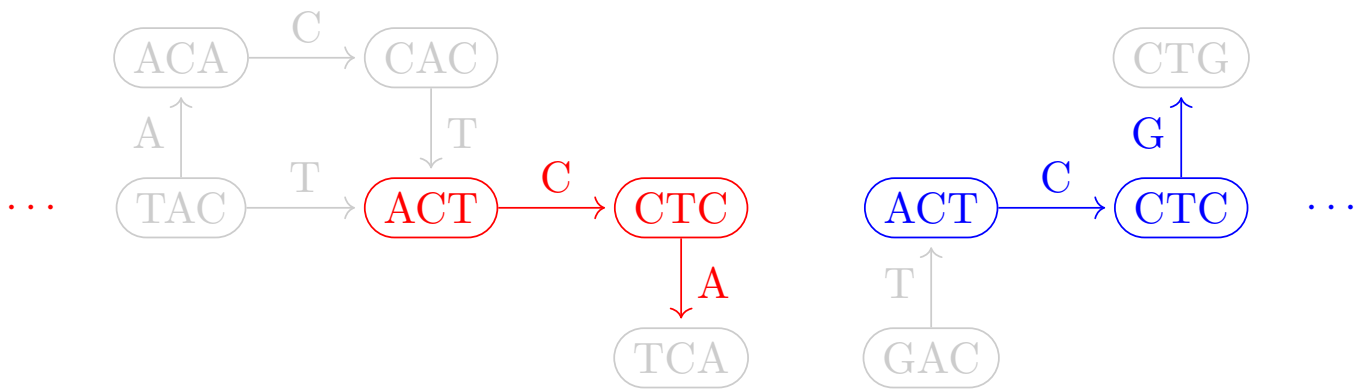
- Tasks:
  1. Merge the nodes in G_0 and G_1 according the order of their k-mers,
  $$\overleftarrow{v_1} \prec \cdots \prec \overleftarrow{v_{n_0}} \qquad \text{and} \qquad \overleftarrow{w_1} \prec \cdots \prec \overleftarrow{w_{n_1}}$$
  2. Recognize when two nodes in G_0 and G_1 refer to the same k-mer, and
  $$\overleftarrow{v_i} \;?{=}\; \overleftarrow{w_j}$$
  3. Properly merge and update W and last.



Merging dBGs

Merging BOSS representations

- Tasks:
  1. Merge the nodes in G_0 and G_1 according the order of their k-mers,
  $$\overleftarrow{v_1} \prec \cdots \prec \overleftarrow{v_{n_0}} \qquad \text{and} \qquad \overleftarrow{w_1} \prec \cdots \prec \overleftarrow{w_{n_1}}$$
  2. Recognize when two nodes in G_0 and G_1 refer to the same k-mer, and
  $$\overleftarrow{v_i} \ ?= \overleftarrow{w_j}$$
  3. Properly merge and update W and last.

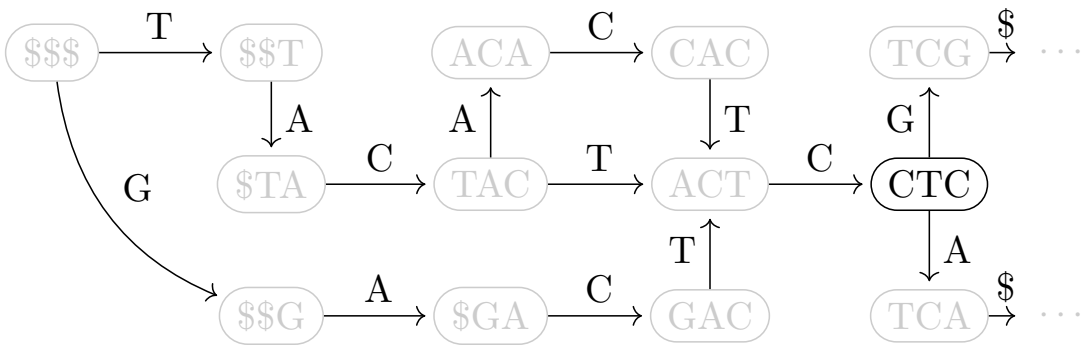


Merging dBGs

1. Merging the nodes in G_0 and G_1:

- The main problem is that in BOSS the k-mers $\overrightarrow{v} = v[1, k]$ are not directly available.
- We will essentially reconstruct them using $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$.

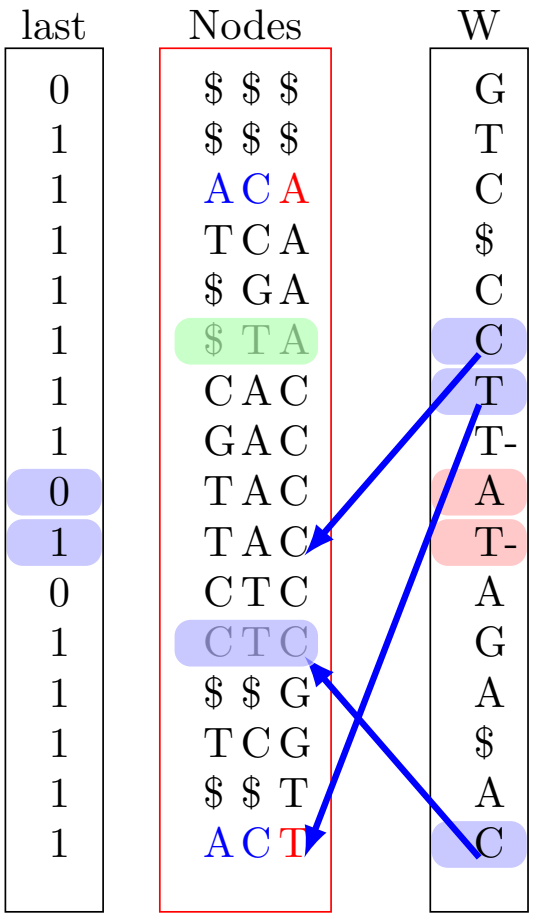
last	Nodes	W
0	\$ \$ \$	G
1	\$ \$ \$	T
1	A C A	C
1	T C A	\$
1	\$ G A	C
1	\$ T A	C
1	C A C	T
1	G A C	T-
0	T A C	A
1	T A C	T-
0	C T C	A
1	C T C	G
1	\$ \$ G	A
1	T C G	\$
1	\$ \$ T	A
1	A C T	C

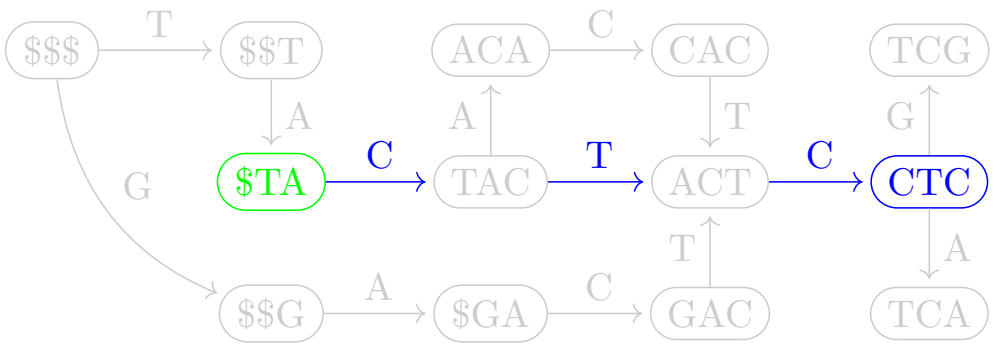


Merging dBGs

1. Merging the nodes in G_0 and G_1:

- The main problem is that in BOSS the k-mers $\overrightarrow{v} = v[1, k]$ are not directly available.
- We will essentially reconstruct them using $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$.

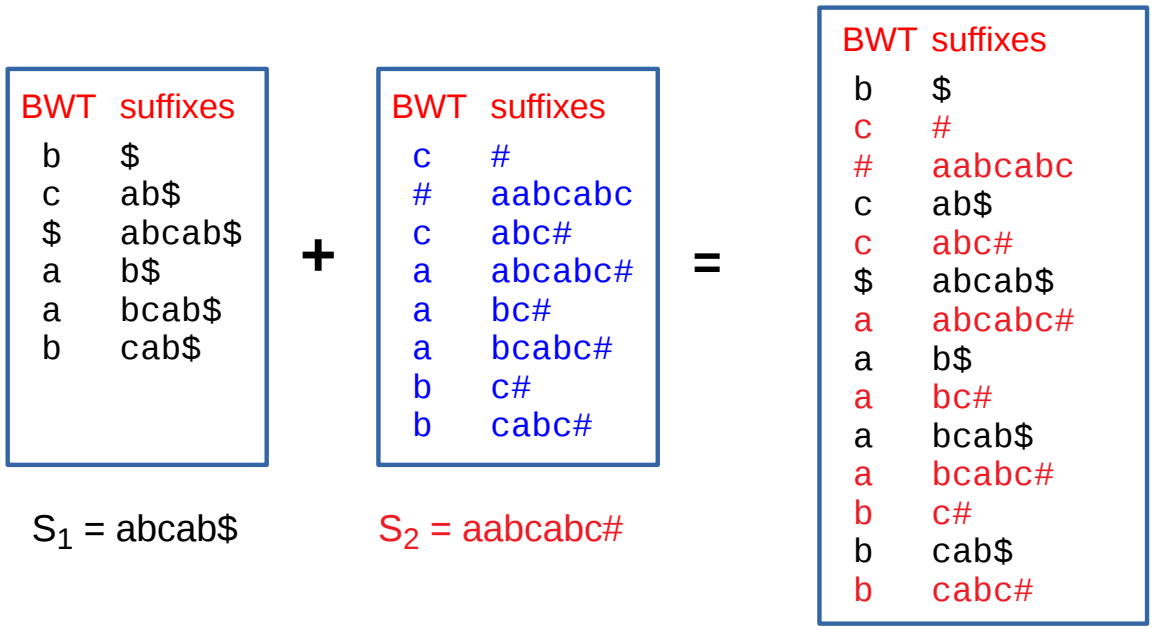




Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.



1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots$, suffixes are h sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$.

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

h=2

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

h=3

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1 h=2 h=3

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
0	c	ab$
0	$	abcab$
1	#	aabcabc
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
0	b	cab$
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1 h=2 h=3

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
0	c	ab$
0	$	abcab$
1	#	aabcabc
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
0	b	cab$
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1 h=2 h=3

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
0	c	ab$
0	$	abcab$
1	#	aabcabc
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
0	b	cab$
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1 h=2 h=3

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
0	c	ab$
0	$	abcab$
1	#	aabcabc
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
0	b	cab$
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1 h=2 h=3

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
0	c	ab$
0	$	abcab$
1	#	aabcabc
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
0	b	cab$
1	b	c#
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1:

- We can merge $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ based on the BWT merging algorithm by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014] .
- Small space: $2n$ bits for Z^{h-1} and Z^h.

h=1

Z	BWT	
0	b	$
0	c	ab$
0	$	abcab$
0	a	b$
0	a	bcab$
0	b	cab$
1	c	#
1	#	aabcabc
1	c	aabc#
1	a	abcabc#
1	a	bc#
1	a	bcabc#
1	b	c#
1	b	cabc#

h=2

Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

h=3

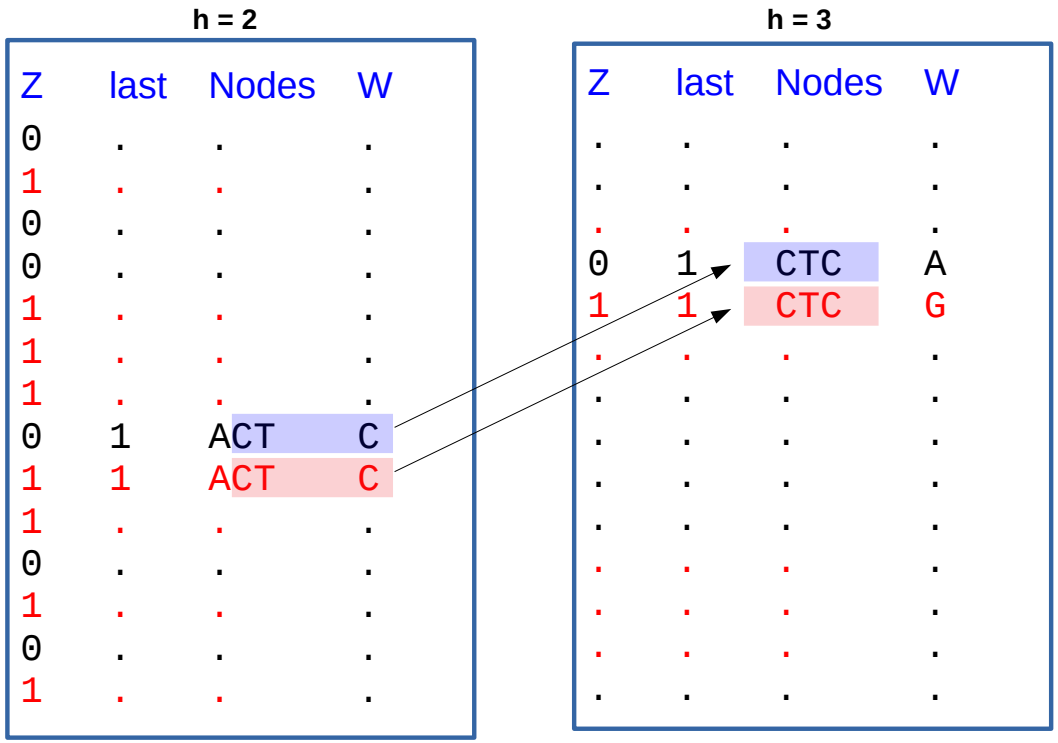
Z	BWT	
0	b	$
1	c	#
1	#	aabcabc
0	c	ab$
0	$	abcab$
1	c	abc#
1	a	abcabc#
0	a	b$
0	a	bcab$
1	a	bc#
1	a	bcabc#
1	b	c#
0	b	cab$
1	b	cabc#

1. Radix sort (BWT and bits); 2. At iteration $h = 1, 2, \ldots,$ suffixes are h-sorted; 3. $\text{BWT}_{01} \leftarrow Z[1, n]$;

Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

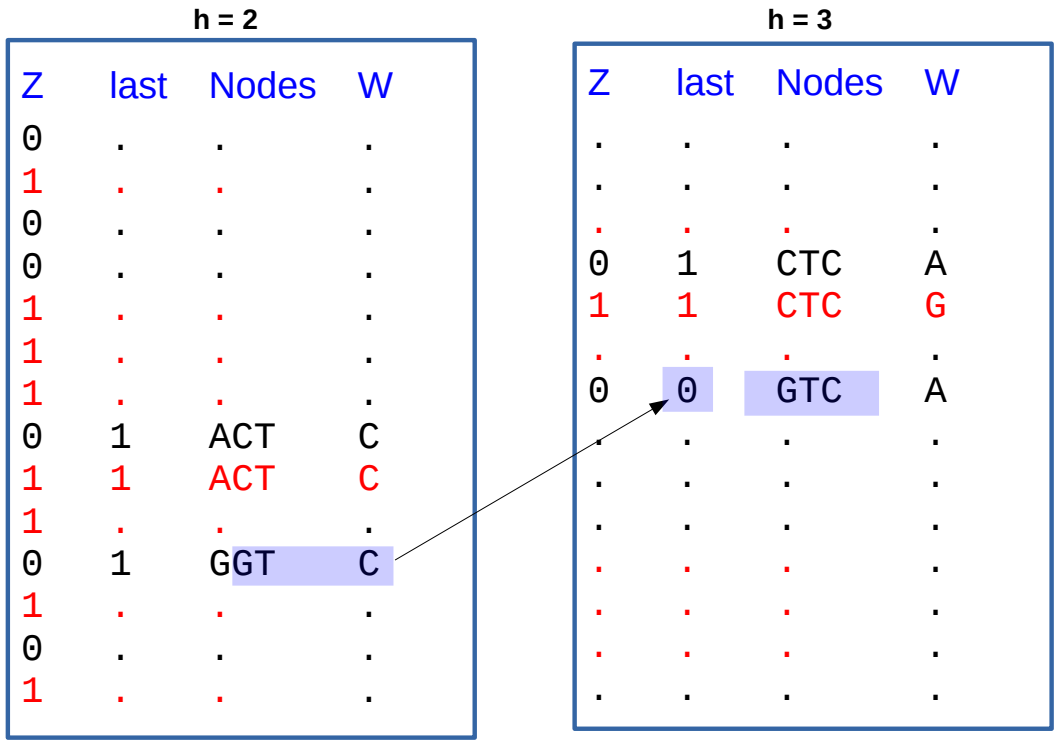
- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

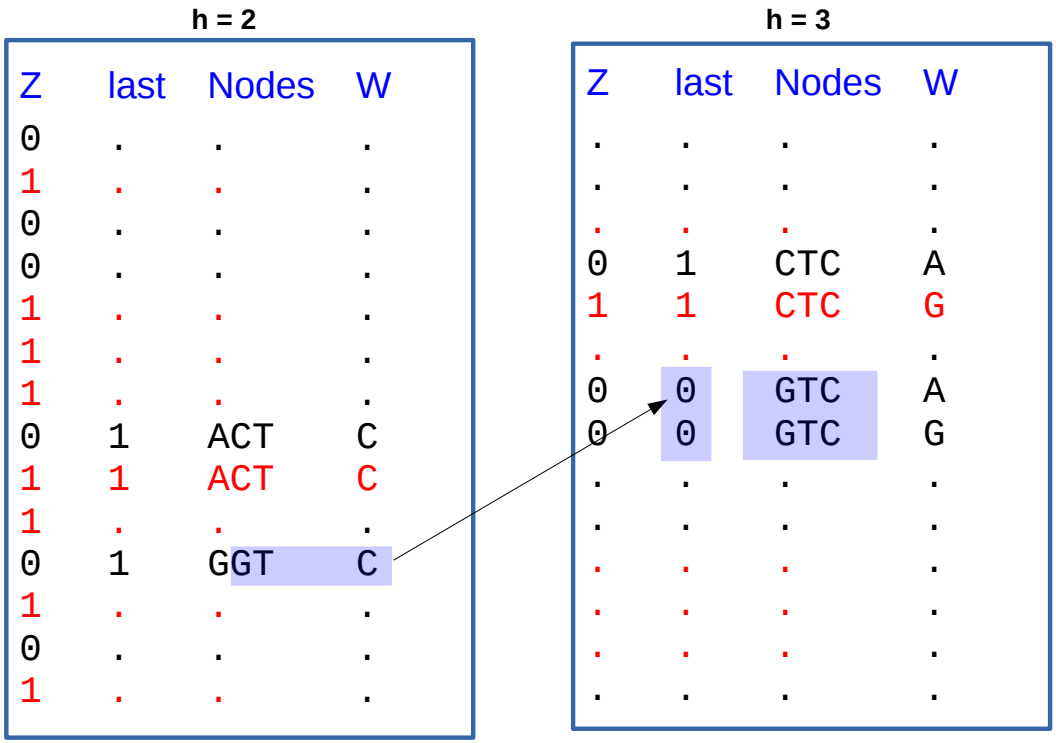
- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

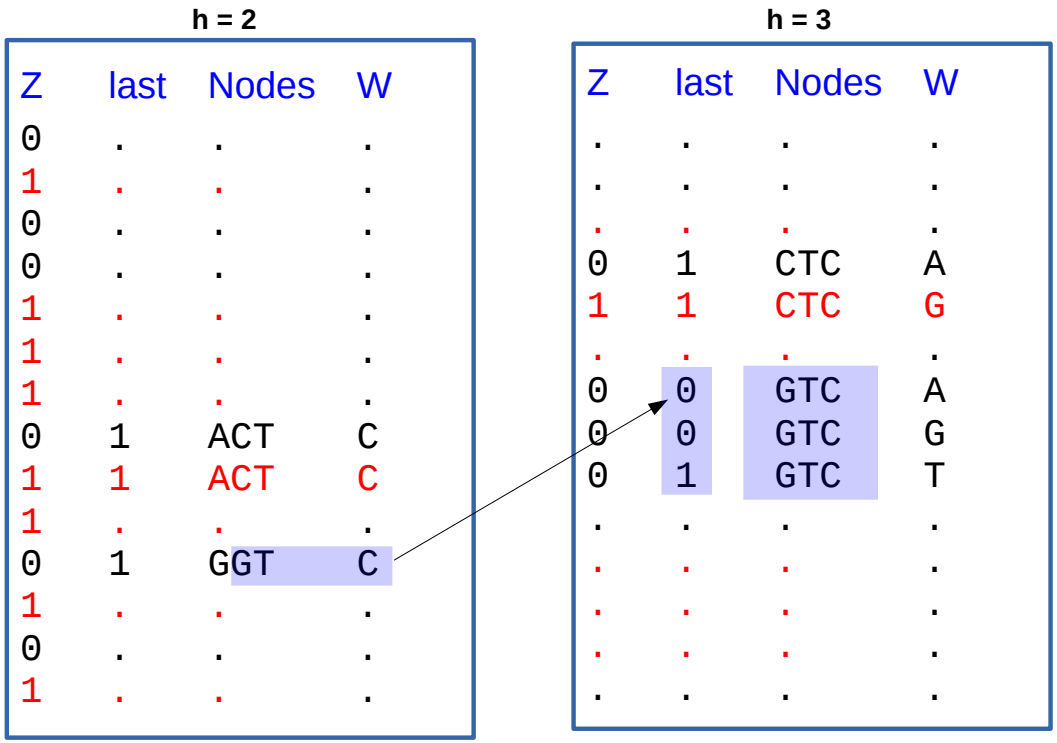
- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

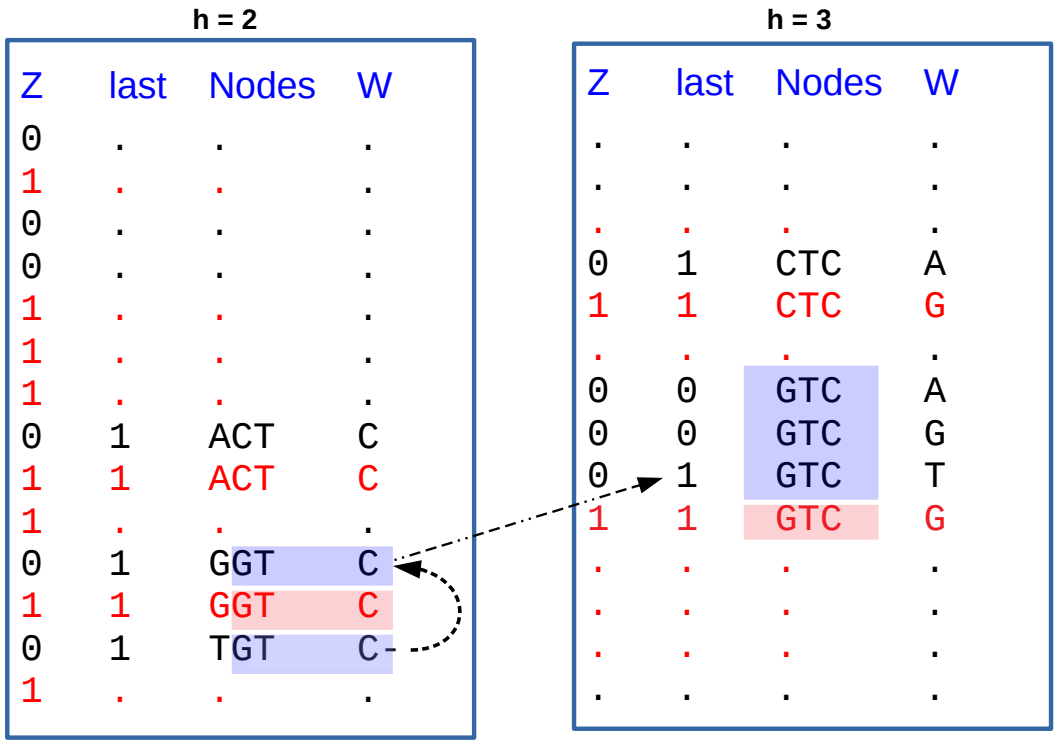
- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

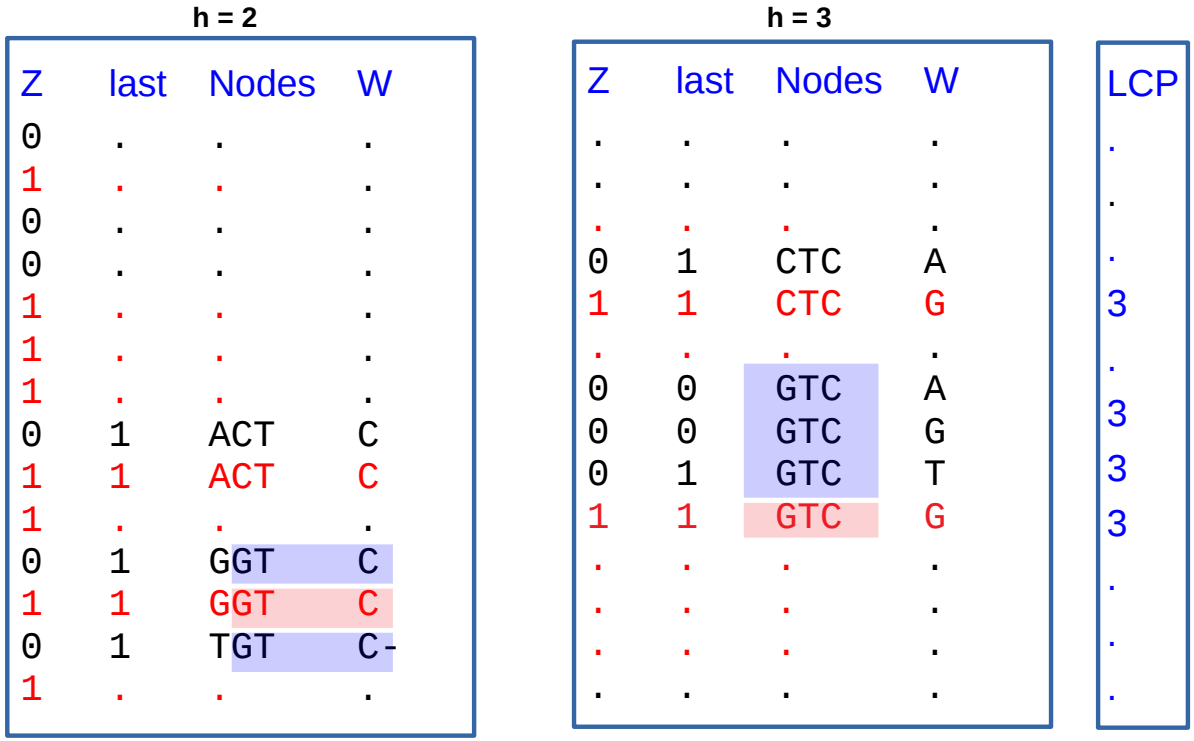
- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



Merging dBGs

1. Merging the nodes in G_0 and G_1: with HM algorithm following the outgoing edges;

- There are two main modifications:
  1. Proceed by blocks according to array $\text{last}_0[1, n_0]$ (resp. $\text{last}_1[1, n_1]$);
  2. Ignore negative symbols.
- We can also compute the LCP array during HM algorithm [Egidi and Manzini, SPIRE 2017]. ← gap algorithm.



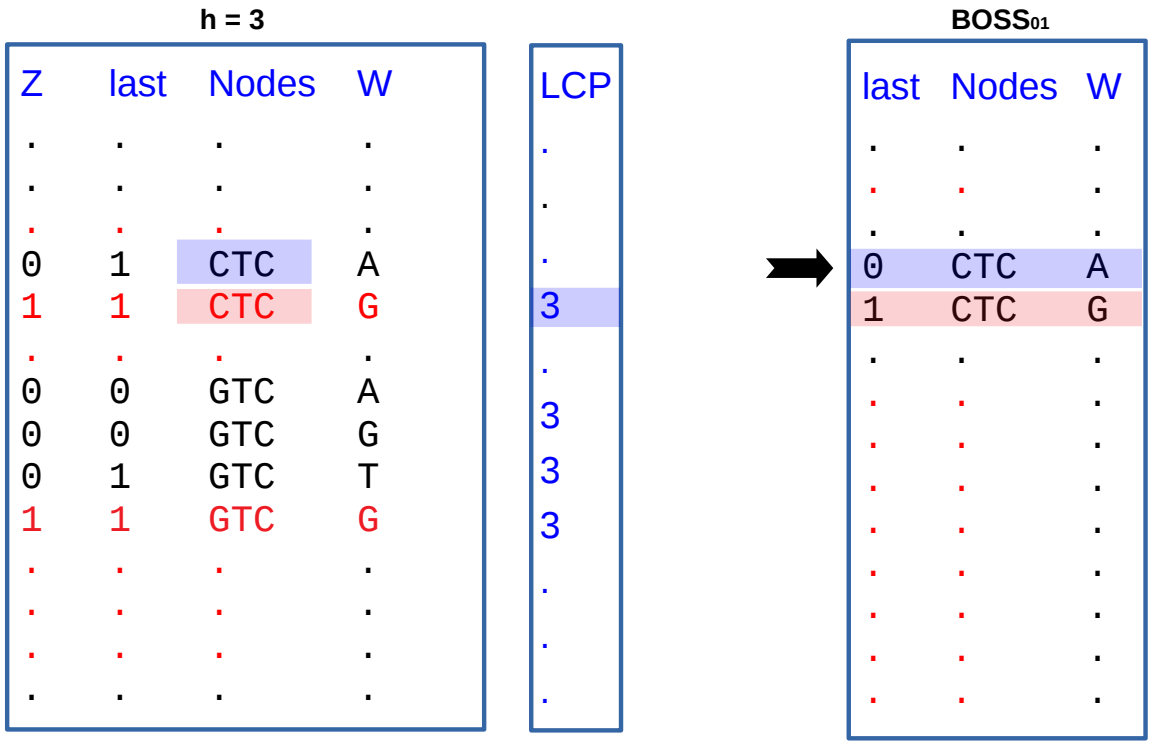
Merging dBGs

2. Recognizing identical k-mers:

- With the LCP array we can decide if $\overleftarrow{v_i}$?= $\overleftarrow{w_j}$.

3. Properly merging $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$:

- $Z[1, n]$ and $\text{LCP}[1, n]$ arrays are enough to merge the sorted nodes.
- Update negative symbols, store the last position $\text{LCP}[i] < k - 1$, and for $c \in \Sigma$ store postion of its $pred[c]$.



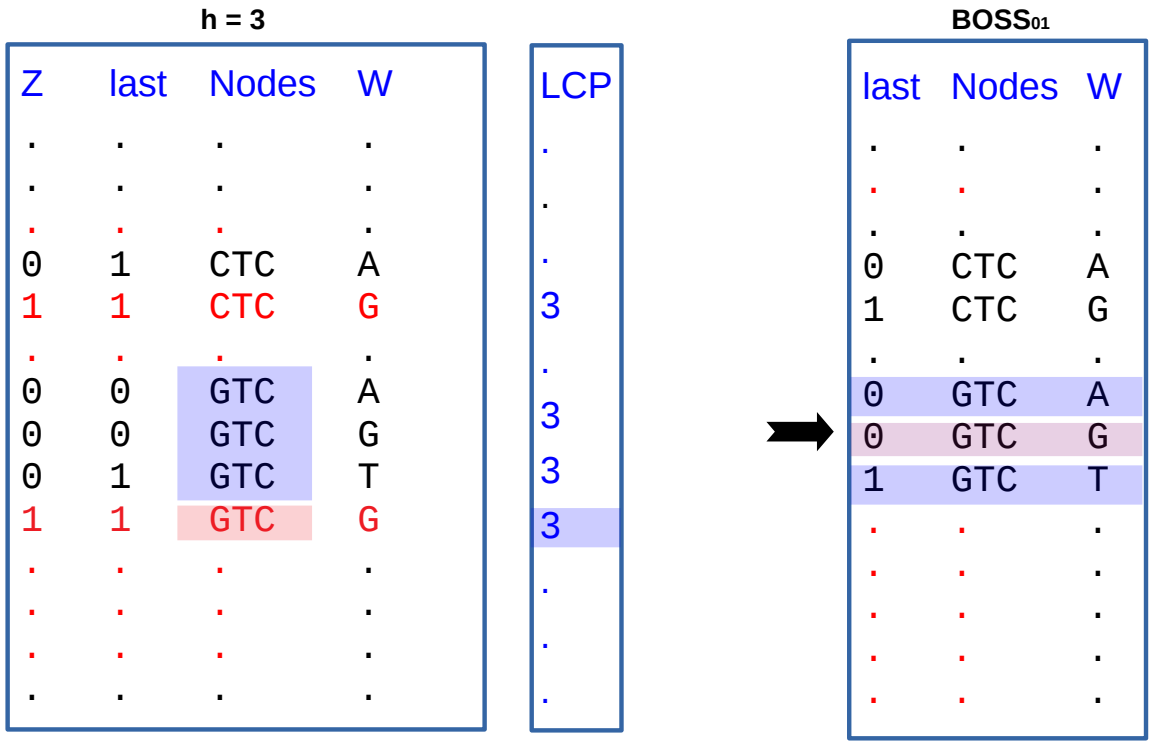
Merging dBGs

2. Recognizing identical k-mers:

- With the LCP array we can decide if $\overleftarrow{v_i}$?= $\overleftarrow{w_j}$.

3. Properly merging $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$:

- $Z[1, n]$ and LCP$[1, n]$ arrays are enough to merge the sorted nodes.
- Update negative symbols, store the last position LCP$[i] < k - 1$, and for $c \in \Sigma$ store postion of its $pred[c]$.



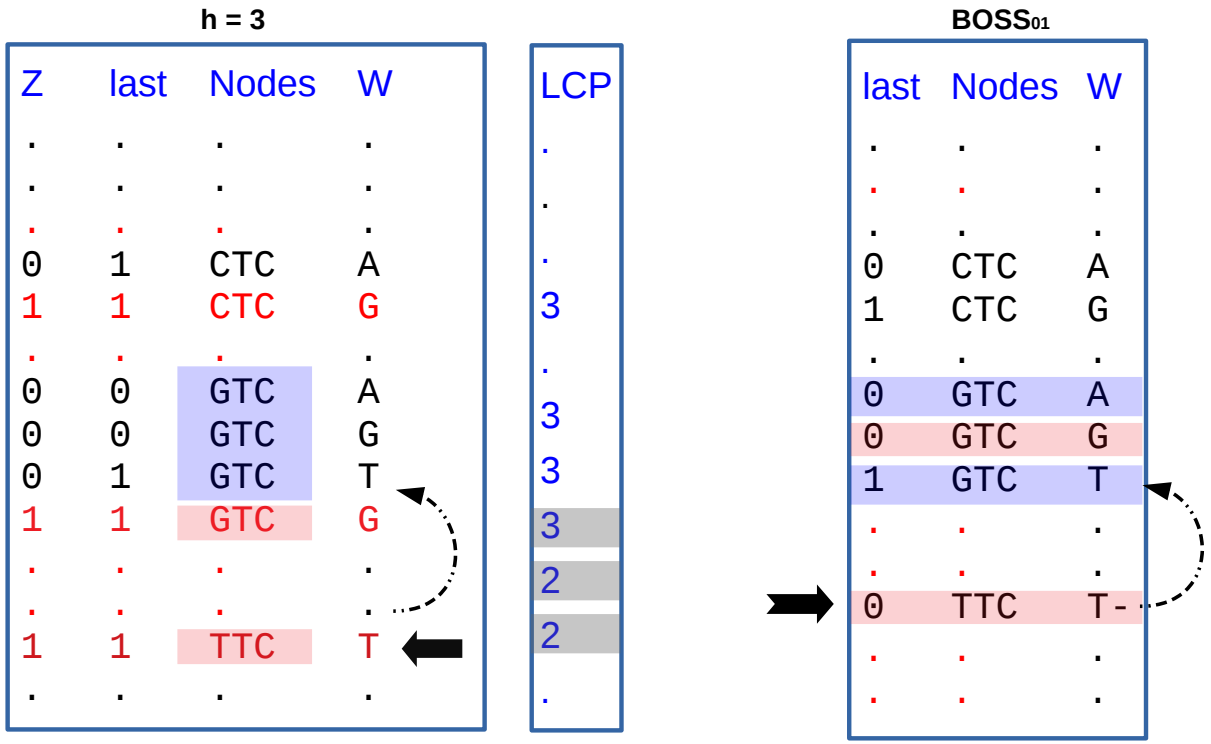
Merging dBGs

2. Recognizing identical k-mers:

- With the LCP array we can decide if $\overleftarrow{v_i}$?= $\overleftarrow{w_j}$.

3. Properly merging $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$:

- $Z[1, n]$ and LCP$[1, n]$ arrays are enough to merge the sorted nodes.
- Update negative symbols, store the last position LCP$[i] < k - 1$, and for $c \in \Sigma$ store postion of its $pred[c]$.



Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$ bits of working space.
  - $2n$ bits for arrays 2^{k-1} and 2^k.
  - Algorithm gap can encode the LCP array into an array of 2-bit ints, we only need to know

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k-1$	10
$\text{LCP}[i] < k-1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$ bits of working space.
  - $2n$ bits for arrays 2^{k-1} and 2^k.
  - Algorithm gap can encode the LCP array into an array of 2-bit cells, we only need to know

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k-1$	10
$\text{LCP}[i] < k-1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$ bits of working space.
  - $2n$ bits for arrays Z^{h-1} and Z^h.
  - Algorithm gap can encode the LCP array bits an array of 2-bits, we only need at most

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k-1$	10
$\text{LCP}[i] < k-1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$-bits of working space.
  - $2n$ bits for arrays Z^{h-1} and Z^h
  - Algorithm gap can encode the LCP array into an array of 2 bits, we only need to know:

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k - 1$	10
$\text{LCP}[i] =< k - 1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$-bits of working space.
  - $2n$ bits for arrays Z^{h-1} and Z^h
  - Algorithm gap can encode the LCP array into an array of 2 bits, we only need to know:

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k - 1$	10
$\text{LCP}[i] =< k - 1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$-bits of working space.
  - $2n$ bits for arrays Z^{h-1} and Z^h
  - Algorithm gap can encode the LCP array into an array of 2 bits, we only need to know:

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k - 1$	10
$\text{LCP}[i] =< k - 1$	01

Merging dBGs

Theoretical costs: $m = m_0 + m_1$ (number of edges), and $n = n_0 + n_1$ (number of nodes).

- Running time: $\mathcal{O}(m + n \cdot k)$-time.
  - We use only positive symbols in Ws: we can make a copy of $\langle W_0, \text{last}_0 \rangle$ and $\langle W_1, \text{last}_1 \rangle$ without negative symbols, total $\mathcal{O}(m)$-time.
  - Each step of gap (HM+LCP) algorithm takes $\mathcal{O}(n)$-time, total $\mathcal{O}(n \cdot k)$-time.
- Space: $4n$-bits of working space.
  - $2n$ bits for arrays Z^{h-1} and Z^h
  - Algorithm gap can encode the LCP array into an array of 2 bits, we only need to know:

value	code
$\text{LCP}[i] = k$	11
$\text{LCP}[i] = k - 1$	10
$\text{LCP}[i] =< k - 1$	01

Merging dBGs

Main result:

The merging of two succinct representations of de Brujin graphs takes $\mathcal{O}(m + n \cdot k)$ time and $4n + \mathcal{O}(1)$ bits of additional space.

Previous result:

Muggli and Boucher [bioRxiv, 2017] showed how to merge de Brujin graphs in $\mathcal{O}(m \cdot k)$ time and $2n(1 + \log \sigma)$ bits of additional space.

Remarks:

- BOSS variations: variable-order and colored.
- Muggli's does not support variable-ordering.
- Most of the data are accessed sequentially ← external memory

1. **Running time** is equivalent for small alphabets, e.g. {A,C,G,T}.
2. **Peak memory** can be prohibitive.

Merging dBGs

Main result:

The merging of two succinct representations of de Brujin graphs takes $\mathcal{O}(m + n \cdot k)$ time and $4n + \mathcal{O}(1)$ bits of additional space.

Previous result:

Muggli and Boucher [bioRxiv, 2017] showed how to merge de Brujin graphs in $\mathcal{O}(m \cdot k)$ time and $2n(1 + \log \sigma)$ bits of additional space.

Remarks:

- BOSS variations: variable-order and colored.
- Muggli's does not support variable-ordering.
- Most of the data are accessed sequentially ← external memory

1. **Running time** is equivalent for small alphabets, e.g. {A,C,G,T}.
2. **Peak memory** can be prohibitive.

Merging dBGs

Main result:

The merging of two succinct representations of de Brujin graphs takes $\mathcal{O}(m + n \cdot k)$ time and $4n + \mathcal{O}(1)$ bits of additional space.

Previous result:

Muggli and Boucher [bioRxiv, 2017] showed how to merge de Brujin graphs in $\mathcal{O}(m \cdot k)$ time and $2n(1 + \log \sigma)$ bits of additional space.

Remarks:

- BOSS variations: variable-order and colored.
- Muggli's does not support variable-ordering.
- Most of the data are accessed sequentially ← external memory

1. **Running time** is equivalent for small alphabets, e.g. {A,C,G,T}.
2. **Peak memory** can be prohibitive.

Merging dBGs

Main result:

The merging of two succinct representations of de Brujin graphs takes $\mathcal{O}(m + n \cdot k)$ time and $4n + \mathcal{O}(1)$ bits of additional space.

Previous result:

Muggli and Boucher [bioRxiv, 2017] showed how to merge de Brujin graphs in $\mathcal{O}(m \cdot k)$ time and $2n(1 + \log \sigma)$ bits of additional space.

Remarks:

- BOSS variations: variable-order and colored.
- Muggli's does not support variable-ordering.
- Most of the data are accessed sequentially ← external memory

1. **Running time** is equivalent for small alphabets, e.g. {A,C,G,T}.
2. **Peak memory** can be prohibitive.

Merging dBGs

Main result:

The merging of two succinct representations of de Brujin graphs takes $\mathcal{O}(m + n \cdot k)$ time and $4n + \mathcal{O}(1)$ bits of additional space.

Previous result:

Muggli and Boucher [bioRxiv, 2017] showed how to merge de Brujin graphs in $\mathcal{O}(m \cdot k)$ time and $2n(1 + \log \sigma)$ bits of additional space.

Remarks:

- BOSS variations: variable-order and colored.
- Muggli's does not support variable-ordering.
- Most of the data are accessed sequentially $\leftarrow$ external memory

1. **Running time** is equivalent for small alphabets, e.g. {A,C,G,T}.
2. **Peak memory** can be prohibitive.

Outline

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, ← compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, $\leftarrow$ compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, $\leftarrow$ compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, $\leftarrow$ compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, $\leftarrow$ compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Space-efficient construction of succinct de Bruijn graphs

Divide-and-conquer:

1. We split $\mathcal{S}$ into smaller subcollections, $\leftarrow$ compute their BWT+LCP array in RAM.
2. For each subcollection: BWT+LCP; then the BOSS representation.
3. Finally, we merge all de Brujin graphs into a single BOSS representation.

Results:

The of de Brujin graph construction for a collection of total length N takes $\mathcal{O}(\log(N/M) \cdot (m + n \cdot k))$ time and $\mathcal{O}(M)$ words and $4n$ bits of RAM.

Remarks:

- We can update (add new graphs) to the de Brujin graph.
- BOSS variations: variable-order and colored.

Obrigado!

Felipe A. Louza
louza@usp.br

[arXiv, 2019]

*Supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP).

Outline


Lavinia Egidi, Felipe Alves Louza, Giovanni Manzini, and Guilherme P. Telles.
External memory BWT and LCP computation for sequence collections with applications.
In *WABI*, volume 113 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.


Lavinia Egidi and Giovanni Manzini.
Lightweight BWT and LCP merging via the Gap algorithm.
In *SPIRE*, volume 10508 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2017.


James Holt and Leonard McMillan.
Constructing Burrows-Wheeler transforms of large string collections via merging.
In *BCB*, pages 464–471. ACM, 2014.


James Holt and Leonard McMillan.
Merging of multi-string BWTs with applications.
Bioinformatics, 30(24):3524–3531, 2014.


Martin D Muggli and Christina Boucher.
Succinct de Bruijn graph construction for massive populations through space-efficient merging.
bioRxiv, 2017.