

TwoPaCo

An efficient algorithm to build the compacted de Bruijn graph
from many complete genomes

Gaspare Ferraro

University of Pisa
Department of Computer Science

Pisa, 29 May 2018



Part I

Introduction



A pan-genomic algorithm

From the greek word *pan* (everything)

Given a series of complete genomes



A pan-genomic algorithm

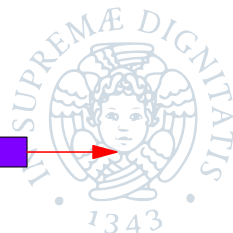
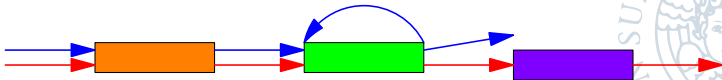
From the greek word *pan* (everything)

Given a series of complete genomes



What we want to see:

- Similarities
- Differences
- Duplications



de Bruijn graph

Sequence #1: **TGACGTC** (2-mers: **TG GA AC CG GT TC**)



Sequence #2: **TGACTTC** (2-mers: **TG GA AC CT TT TC**)



de Bruijn graph

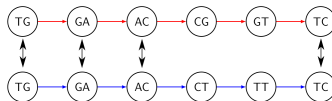
Sequence #1: **TGACGTC** (2-mers: **TG GA AC CG GT TC**)



Sequence #2: **TGACTTC** (2-mers: **TG GA AC CT TT TC**)



Find common vertices



de Bruijn graph

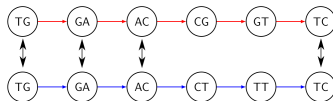
Sequence #1: **TGACGTC** (2-mers: **TG GA AC CG GT TC**)



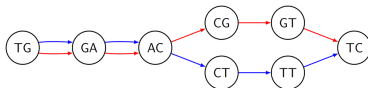
Sequence #2: **TGACTTC** (2-mers: **TG GA AC CT TT TC**)



Find common vertices



Merge them



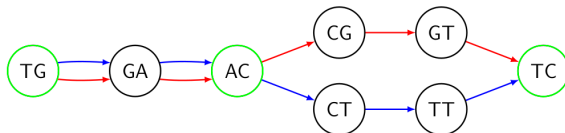
Compacted de Bruijn graph

To compact the de Bruijn graph we need to find the junctions

A vertex is a junction if :

Is the initial or final part of some sequence

Has more than 1 in-edge or more than 1 out-edge



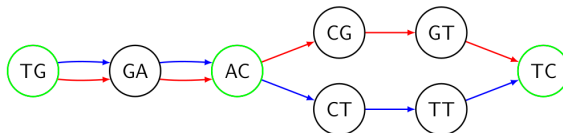
Compacted de Bruijn graph

To compact the de Bruijn graph we need to find the junctions

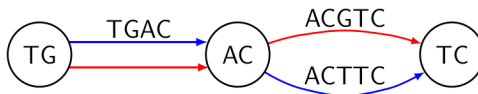
A vertex is a junction if :

Is the initial or final part of some sequence

Has more than 1 in-edge or more than 1 out-edge



Remove simple paths by connect junctions



The problem

How to construct the compacted de Bruijn graph,
from many complete genomes,
avoiding the ordinary graph construction?



The problem

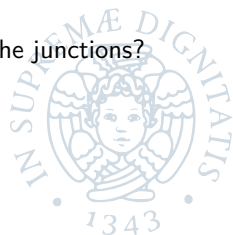
How to construct the compacted de Bruijn graph,
from many complete genomes,
avoiding the ordinary graph construction?



Which k -mers are junctions?



How to construct the compacted de Bruijn graph given the junctions?



Part II

The algorithm



Naive algorithm

- Store all $(k + 1)$ -mers in a hash table
- For each k -mers query the possible edges
- If only 1 in and 1 out edge, unmark as a junction

Algorithm 1: FILTER-JUNCTIONS

Input : $S = \{s_1, \dots, s_m\}$ genome sequences

k integer, size of k -mers

E empty set data structure

C Candidate set of junctions (naively all positions are marked)

Output: A reduce candidate set of junctions C

```

1 foreach  $s \in S$  do
2   for  $1 \leq i < |s| - k$  do
3     if  $C[s, i] = \text{marked}$  then
4        $E \leftarrow E \cup s[i..i+k] \cup s[i-1..i+k-1]$   $\triangleright$  Store all  $(k+1)$ -mers

5 foreach  $s \in S$  do
6   for  $1 \leq i < |s| - k$  do
7     if  $C[s, i] = \text{marked}$  then
8        $(in, out) \leftarrow (0, 0)$   $\triangleright$  Count in/out edges
9       foreach  $c \in \{A, C, G, T\}$  do
10        if  $v \cdot c \in E$  then
11           $in \leftarrow in + 1$ 
12        if  $c \cdot v \in E$  then
13           $out \leftarrow out + 1$ 
14        if  $(in, out) = (1, 1)$  then
15           $C[s, i] = \text{unmarked}$   $\triangleright$  surely not a junction

16 return  $C$ 

```



The memory issue

First part of the naive algorithm:

```
foreach  $s \in S$  do  
  for  $1 \leq i < |s| - k$  do  
    if  $C[s, i] = \textit{marked}$  then  
       $E \leftarrow E \cup s[i..i + k] \cup s[i - 1..i + k - 1]$ 
```



The memory issue

First part of the naive algorithm:

```

foreach  $s \in S$  do
  for  $1 \leq i < |s| - k$  do
    if  $C[s, i] = \textit{marked}$  then
       $E \leftarrow E \cup s[i..i + k] \cup s[i - 1..i + k - 1]$ 

```

We don't really need and, in almost all practical cases,
we can't store all the possible $(k + 1)$ -mers.

Mainly because **only a little percentual** of them are
junction in the de Bruijn graph.



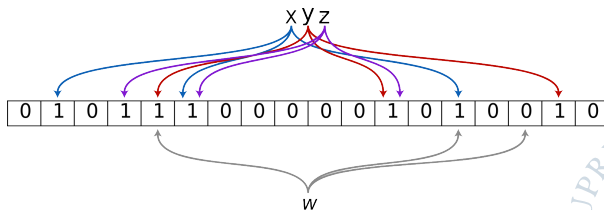
Bloom filter

A space-efficient probabilistic hash table

Bitmap V of size b , h hash functions $f_0, f_1, \dots, f_{h-1} : U \rightarrow [0, b-1]$

insertion(x) $\rightarrow V[f_i(x)] = 1, \forall 0 \leq i < h$

contains(x) \rightarrow **probably yes** if $V[f_i(x)] = 1, \forall 0 \leq i < h$



Probability of false positive, after n insertion: $p_{FP} \simeq (1 - e^{-hn/m})^h$

Two Pass version

- First pass: Select a set of junction candidates by insert all the $(k + 1)$ -mers in a bloom filter of choosing size
- Second pass: Filter out the false positive by storing the reduce sets of $(k + 1)$ -mers in an hash table

Algorithm 2: FILTER-JUNCTIONS-TWO-PASS

Input : strings $S = \{s_1, \dots, s_m\}$ genome sequences

integer k , size of k -mers

integer b , size of bloom filter

Candidate set of junctions C_{in} (naively all positions are marked)

Output: A reduce candidate set of junctions C_{out}

- $F \leftarrow$ empty bloom filter of size b ▷ First pass
 - $C_{temp} \leftarrow \text{FILTER-JUNCTIONS}(S, k, F, C_{in})$
 - $H \leftarrow$ empty hash table ▷ Second pass
 - $C_{out} \leftarrow \text{FILTER-JUNCTIONS}(S, k, H, C_{in})$
 - return** C_{out}
-

The memory issue²

How much memory do we use now?

- First pass: Bloom filter of size b (of our decision)
- Second pass: Hash table containing $(k + 1)$ -mers of junction candidates

We don't know the possible size of the hash table in the second pass.
What if the hash table is not small enough?



The memory issue²

How much memory do we use now?

- First pass: Bloom filter of size b (of our decision)
- Second pass: Hash table containing $(k + 1)$ -mers of junction candidates

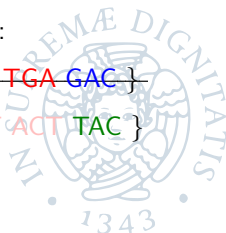
We don't know the possible size of the hash table in the second pass.
What if the hash table is not small enough?

Solution: Split the input k -mers in chunks and analyze them in multiple rounds.

Split *TGACTACTGAC* in 3 chunks of 3-mers:

~~Wrong: { TGA GAC ACT }, { CTA TAC ACT }, { CTG TGA GAC }~~

Correct: { TGA TGA CTG }, { GAC GAC CTA }, { ACT ACT TAC }



k -mers splitting

- Count how many k -mers are mapped in each of the c hash value ($c = 2^l$)
- Iterate over all the hash value and insert them in the current chunk
- If the size of current chunks exceeds the average pass to the next chunk

Algorithm 3: ROUND-SPLITTING

Input : strings $S = \{s_1, \dots, s_m\}$ genome sequences

integer k , size of k -mers

integer b , size of bloom filter

integer l , number of rounds

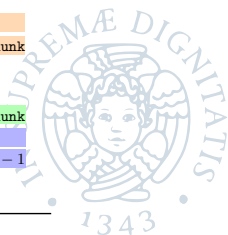
function $f(x)$, from k -mers to integers

Output: $(V_0, V_1, \dots, V_{l-1})$ chunks of k -mers

```

1  $V_0 \leftarrow \emptyset, V_1 \leftarrow \emptyset, \dots, V_{l-1} \leftarrow \emptyset$  ▷ Chunks, initially empty
2  $c_0 \leftarrow 0, c_1 \leftarrow 0, \dots, c_{q-1} \leftarrow 0$  ▷ Counters, for balancing  $k$ -mers
3  $c_i \leftarrow |\{k\text{-mer } x \mid f(x) = i\}|$ 
4  $T \leftarrow \sum_{0 \leq t < q} c_t / l$  ▷ Average size of a chunk
5  $idx \leftarrow 0$ 
6 for  $0 \leq i < q$  do
7    $V_{idx} = V_{idx} \cup \{i\}$  ▷ Add  $i$  to the current chunk
8   if  $|V_{idx}| \geq T$  then
9      $idx \leftarrow \min(idx + 1, l - 1)$  ▷ Next chunk, until reach  $l - 1$ 
10 return  $(V_0, V_1, \dots, V_{l-1})$ 

```



Multiple rounds: dealing with memory restrictions

- Partitionate the input with ROUND-SPLITTING
- Analyse each partition with FILTER-JUNCTIONS-TWO-PASS
- Merge the results of each rounds and output the real junctions

Algorithm 4: TWO-PaCo

Input : strings $S = \{s_1, \dots, s_m\}$ genoma sequences

integer k , size of k -mers

integer b , size of bloom filter

integer l , number of rounds

function $f(x)$, from k -mers to integers

Output: C_{final} all the junctions in the compacted de Bruijn graph

```

1  $(V_0, V_1, \dots, V_{l-1}) \leftarrow \text{ROUND-SPLITTING}(S, k, b, l, f)$  ▷ Create chunks
2  $C_{init} \leftarrow$  Boolean array with every position unmarked
3 for  $0 \leq i < l$  ▷ For each chunks
4 do
5    $C_i \leftarrow$  mark every position of  $C_{init}$  that starts a  $k$ -mer with hash in  $V_i$ 
6    $C_i \leftarrow \text{FILTER-JUNCTIONS-TWO-PASS}(S, k, b, C_i)$  ▷ Find junctions
7  $C_{final} = \bigcup C_i$  ▷ Merge junctions of each chunks
8 return  $C_{final}$ 

```

Parallelization scheme

As far we focused in reducing memory. What about the time?



Parallelization scheme

As far we focused in reducing memory. What about the time?

TwoPaCo can be easily parallelizable:

- Parallel for loops (OpenMP, TBB, ...)
- Concurrent bloom filter (parallel insert and search)
- Concurrent hash table (parallel insert and search)



Parallelization scheme

As far we focused in reducing memory. What about the time?

TwoPaCo can be easily parallelizable:

- Parallel for loops (OpenMP, TBB, ...)
- Concurrent bloom filter (parallel insert and search)
- Concurrent hash table (parallel insert and search)

We made a clever usage of the concurrent data structs:

- Only insertion in the first part
- Only query in the second part (static table, no race conditions)



Part III

Results



Source code & Dataset

Original source code by medvedev group available on:

<https://github.com/medvedevgroup/TwoPaCo>

Personal implementation and presentation available on:

<https://github.com/GaspereG/TwoPaCo>

<https://github.com/GaspereG/TwoPaCoPresentation>

Dataset for experiments:

- 62 Escherichia coli (~300Mb)
- 5 humans (~21Gb)
- 8 primates (~23Gb)
- 100 simulated human (~400Gb)



Memory complexity

The memory complexity is the maximum among the first and the second pass of TwoPaCo

- First pass: insert all k -mers in a bloom filter of **size** b
- Second pass: store all **junction candidates** in a hash table



Memory complexity

The memory complexity is the maximum among the first and the second pass of TwoPaCo

- First pass: insert all k -mers in a bloom filter of **size** b
- Second pass: store all **junction candidates** in a hash table

How many junction candidates?

- Real junction, J
- False positive induced from the bloom filter, FP



Memory complexity

The memory complexity is the maximum among the first and the second pass of TwoPaCo

- First pass: insert all k -mers in a bloom filter of **size** b
- Second pass: store all **junction candidates** in a hash table

How many junction candidates?

- Real junction, J
- False positive induced from the bloom filter, FP

Result: $\mathcal{O}(\max\{b, (J + FP)k\})$



Time complexity

The time complexity is the sum
between the first and the second pass of TwoPaCo

- First pass: **insert all k -mers** in a bloom filter using h hash functions
- Second pass: iterate over all **candidate positions** and query the hash table



Time complexity

The time complexity is the sum between the first and the second pass of TwoPaCo

- First pass: **insert all k -mers** in a bloom filter using h hash functions
- Second pass: iterate over all **candidate positions** and query the hash table

How many k -mers?

$\mathcal{O}(m)$, where $m = \sum_{s \in S} |s|$ is the total input size



Time complexity

The time complexity is the sum between the first and the second pass of TwoPaCo

- First pass: **insert all k -mers** in a bloom filter using h hash functions
- Second pass: iterate over all **candidate positions** and query the hash table

How many k -mers?

$\mathcal{O}(m)$, where $m = \sum_{s \in S} |s|$ is the total input size

How many candidate positions?

- Real positions, $|G_c|$
- False positive induced from the bloom filter, FP



Time complexity

The time complexity is the sum between the first and the second pass of TwoPaCo

- First pass: **insert all k -mers** in a bloom filter using h hash functions
- Second pass: iterate over all **candidate positions** and query the hash table

How many k -mers?

$\mathcal{O}(m)$, where $m = \sum_{s \in S} |s|$ is the total input size

How many candidate positions?

- Real positions, $|G_c|$
- False positive induced from the bloom filter, FP

Result: $\mathcal{O}(\textcolor{green}{mh} + (|G_c| + \textcolor{brown}{FP})k)$



Complexity comparison

State of the art for compressed de Bruijn graph construction:

- Sibelia (Minkin, Patel, Kolmogorov, Vyahhi, Pham, 2013)
- SplitMEM (Marcus, Lee, Schatz, 2014)
- bwt-based (Baier, Beller, Ohlebusch, 2015)



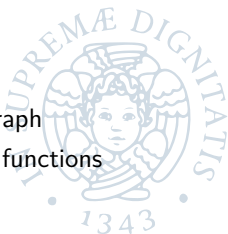
Complexity comparison

State of the art for compressed de Bruijn graph construction:

- Sibelia (Minkin, Patel, Kolmogorov, Vyahhi, Pham, 2013)
- SplitMEM (Marcus, Lee, Schatz, 2014)
- bwt-based (Baier, Beller, Ohlebusch, 2015)

Algorithm	Time complexity	Memory complexity
Sibelia	$\mathcal{O}(m)$	$\mathcal{O}(m)$
SplitMEM	$\mathcal{O}(m \log g)$	$\mathcal{O}(G_c + m)$
bwt-based	$\mathcal{O}(m)$	$\mathcal{O}(m)$
TwoPaCo	$\mathcal{O}(mh + (G_c + FP)k)$	$\mathcal{O}(\max\{b, (J + FP)k\})$

- $m = \sum_{s \in S} |s|$, total input size
- $g = \max_{s \in S} |s|$, size of the biggest genoma
- J and $|G_c|$, number of vertex and edge in the de Bruijn Graph
- b and h , size of the bloom filter table and number of hash functions
- FP , number of false positives in first pass



Running time comparison

What are the practical performances of TwoPaCo against the state of the art?



Running time comparison

What are the practical performances of TwoPaCo against the state of the art?

Dataset	Sibelia	SplitMEM	bwt-based	TwoPaCo	
				1 thread	15 thread
62 E.coli (k=25)	10 (12.2)	70 (178.0)	8 (0.85)	4 (0.16)	2 (0.39)
62 E.coli (k=100)	8 (7.6)	67 (178.0)	8 (0.50)	4 (0.19)	2 (0.39)
7 humans (k=25)	-	-	867 (100.30)	436 (4.40)	63 (4.84)
7 humans (k=100)	-	-	807 (46.02)	317 (8.42)	57 (8.75)
8 primates (k=25)	-	-	-	914 (34.36)	111 (34.36)
8 primates (k=100)	-	-	-	756 (56.06)	101 (61.68)
50 humans (k=25)	-	-	-	-	705 (69.77)
50 humans (k=100)	-	-	-	-	927 (70.21)

Running times in minutes and memory usage in gigabytes in parenthesis

- - on Sibelia = out of time
- - on SplitMEM = out of memory
- - on bwt-based = out of memory
- - on TwoPaCo = experiment not done



Fixed memory

In how many rounds do we need to split the input to not exceeding a given memory threshold?



Fixed memory

In how many rounds do we need to split the input to not exceeding a given memory threshold?

Threshold	Used memory	Bloom filter size	Running time	Rounds
10GB	8.62GB	8.59GB (2^{33})	4h	1
8GB	6.73GB	4.29GB (2^{32})	7h	3
6GB	5.98GB	4.29GB (2^{32})	9h	4
4GB	3.51GB	2.14GB (2^{31})	11h	6

Experiments on 5 simulated human genomes, $k = 25$, 8 threads



Fixed memory

In how many rounds do we need to split the input to not exceeding a given memory threshold?

Threshold	Used memory	Bloom filter size	Running time	Rounds
10GB	8.62GB	8.59GB (2^{33})	4h	1
8GB	6.73GB	4.29GB (2^{32})	7h	3
6GB	5.98GB	4.29GB (2^{32})	9h	4
4GB	3.51GB	2.14GB (2^{31})	11h	6

Experiments on 5 simulated human genomes, $k = 25$, 8 threads

Result: **we can trade-off memory for time**



Bloom Filter false positive

Is the Bloom Filter really efficient in reducing junction candidates?



Bloom Filter false positive

Is the Bloom Filter really efficient in reducing junction candidates?

Dataset	k	Junction candidates		
		Initial	First pass	Second pass
62 E.coli	25	310 157 564 (100%)	24 649 489 (7.94%)	24 572 562 (7.92%)
62 E.coli	100	310 157 489 (100%)	22 848 018 (7.36%)	9 492 091 (3.06%)
7 humans	25	21 201 290 922 (100%)	3 489 946 013 (16.46%)	2 974 098 154 (14.02%)
7 humans	100	21 201 290 847 (100%)	1 374 287 870 (6.48%)	188 224 214 (0.88%)
8 primates	25	24 540 556 921 (100%)	5 423 003 377 (22.09%)	5 401 587 503 (22.01%)
8 primates	100	24 540 556 846 (100%)	1 174 160 336 (4.78%)	502 441 107 (2.04%)

- Initial: total number of k -mers in dataset
- First pass: number of junction candidates (using a bloom filter)
- Second pass: real number of junction (using an hash table)



Conclusion

- Lowest time/memory usage in the state of the art
- Highly scalable (best performance with multi-thread CPUs)
- Time/memory trade-off (suitable for small computer)
- Bloom filter perform very well in practice



References

[1] Minkin, I., Pham, S., Medvedev, P. (2016).

TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes

[2] Minkin, I., Patel, A., Kolmogorov, M., Vyahhi, N., Pham, S. (2013).

Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes.

[3] Marcus, S., Lee, H., Schatz, M. C. (2014).

SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips.

[4] Baier, U., Beller, T., Ohlebusch, E. (2015).

Graphical pan-genome analysis with compressed suffix trees and the **Burrows-Wheeler transform**.



Questions

Thanks for your attention!

Questions?

