University of Stuttgart
Institute of Biomedical Genetics
RNA Biology and Bioinformatics

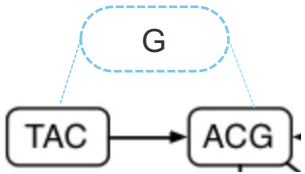
Succinct de Bruijn graphs

Alex Bowe, Taku Onodera, Kunihiko Sadakane
and Tetsuyo Shibuya

De Bruijn Graph

- De Bruijn Graph – DBG
  - cool data structure for DNA sequencing
  - pre-step in DNA assembly
- “De Bruijnizing” algorithm
  1. Chop the reads into **k-mers** (strings of length *k*)
  2. Build the **directed graph** based on the overlaps
  3. Call the k-mers **nodes**
  4. Call the overlaps **edges**

```
TACGACGTCGACT:
TAC
 ACG
  CGA
   GAC
                etc
```

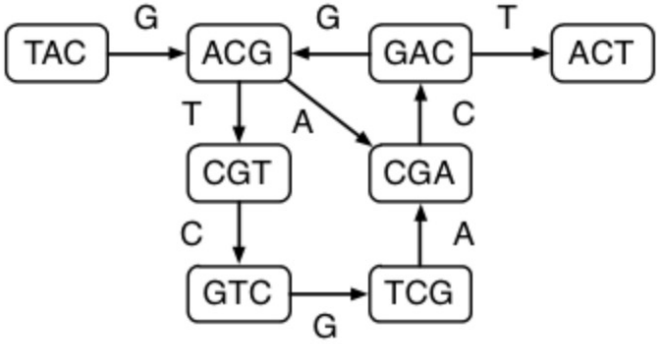


Storage of DBG

- How would you normally store DBG
  - Naive implementation: `dictionary`
    - Pros: Can be easily controlled/efficient **query operations**
    - Cons: **A lot of space** especially in case of metagenomics
- Optimize space

  ❑Rules:
  - Keep efficient query operations
  - Compression allowed while saving the k-mers

  ✓Solution:
  - Information-theoretic lower bound - Z
  - Or close to Z (let me explain)



From	To
TAC	ACG
ACG	CGT, CGA
CGT	GTC
GTC	TCG
TCG	CGA
CGA	GAC
GAC	ACT

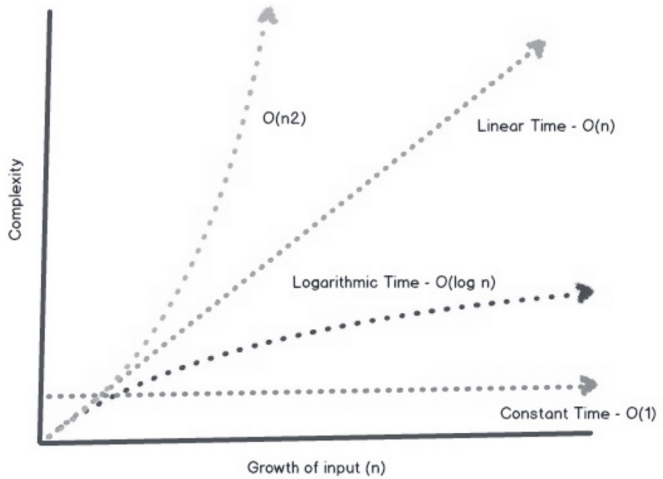
Information Theoretic Lower Bound - OPT

- Given: information-theoretic lower bound: Z
- 3 ways to keep efficient query operations:
  - Implicit: $Z + \mathcal{O}(1)$ bits
  - **Succinct: $Z + o(Z)$ bits**
  - Compact: $\mathcal{O}(Z)$ bits
- *Succinct data structures:*
  - use space “close” to theoretical lower bound
  - allow efficient query operations
  - with efficient “dynamicity” also allow addition/removal of nodes

Information Theoretic Lower Bound:
Round 6.449 to one character after .
Rule: Max rounding would be 0.099.

$$Z = 6.4$$

$o(Z)$ unit is any function that grows slower than a linear function, e.g.: $\sqrt{Z}$, $lg(Z)$, etc.



Succinct de Bruijn Graphs

Alexander Bowe[1], Taku Onodera[2], Kunihiko Sadakane[1], and Tetsuo Shibuya[2]

[1] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku,
Tokyo 101-8430, Japan
{alex,sada}@nii.ac.jp
[2] Human Genome Center, Institute of Medical Science,
University of Tokyo 4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan
{tk-ono,tshibuya}@hgc.jp

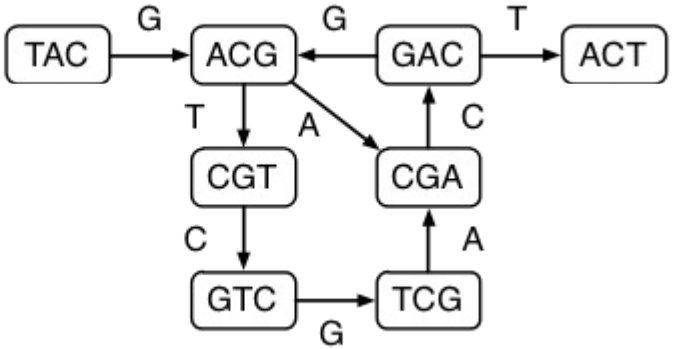
SDBG – Succinct de Bruijn Graph

- Goal:
  - represent de Bruijn graph succinctly $Z + o(Z)$
  - enable efficient query operations
- Succinct de Bruijn graph:
  - DNA sequence of length: N
  - number of edges: m
  - $4m + o(m)$
- Comparisons:
  - distributed hash table(compressed dictionary): 336 GB
  - sparse bitvector representation:
    - $\mathcal{O}(mk)$ w/ encoding and compression: 32 GB
  - ***succinct representation***: 2.5 GB

Construction

- Take every <*node,edge*>
- Sort them based on reverse(colex order) of node label
- Remove duplicates

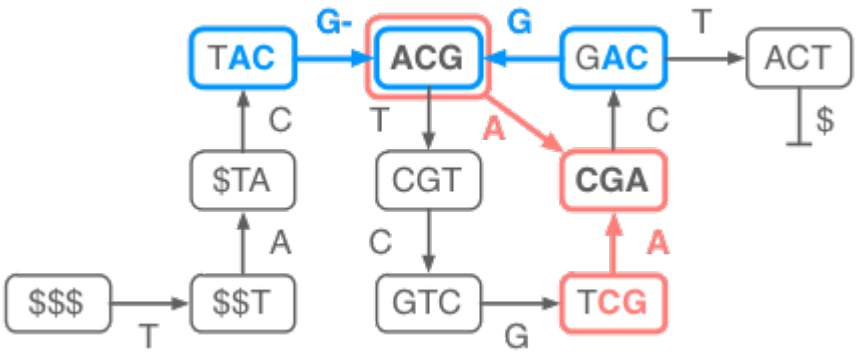
- Additional steps:
  - Use padding to know the start and end
  - Rules for padding:
    - k "$"s to the beginning
    - 1 to the end

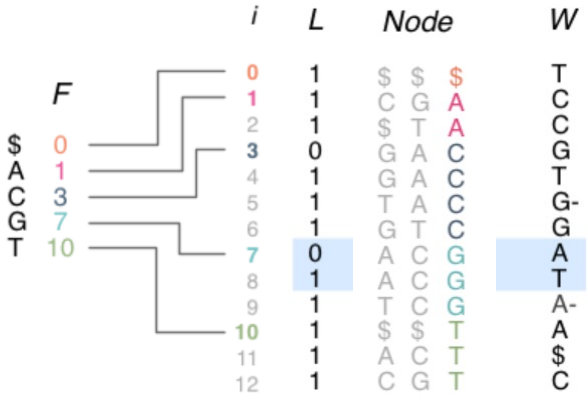


Node	*W*
C G A	C
G A C	G
G A C	T
T A C	G-
G T C	G
A C G	A
A C G	T
T C G	A-
C G T	C

Construction

- $\boldsymbol{W}$ – edge vector(1 char)
- disambiguate identically labelled incoming edges:
  - A and A- , G and G-
- each outgoing edge is stored contiguously
- Bit-vector $\boldsymbol{L}$:
  - introduce a bit vector($0\ or\ 1$):
  - represent whether an edge is the *last* edge exiting a node
- Bit-vector $\boldsymbol{F}$:
  - indicates the first and last occurrences of characters from alphabet $\{\$, A, C, G, T\}$
  - works due to colex ordering

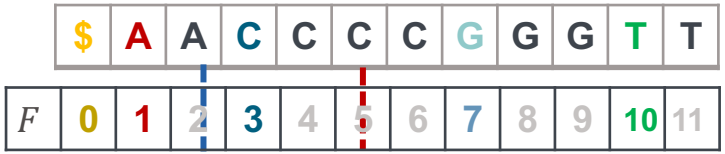




Rank and Select

RANK and SELECT have been used used before on binary alphabets.

RANK



- Returns the number of elements equal to q up to position x

$$rank_q(x) = |\{l, l \in [0 \dots x], S[l] = q\}|$$

$$rank_A(7) = 2$$

(num of **A**'s are there until position 7)

$$rank_C(5) = 3$$

(num of **C**'s are there until position 5)

SELECT

- Returns the position of the x^{th} occurunce of q.

$$select_q(x) = min\{l \in [0 \dots x] : rankq_{(l)} = x\}$$

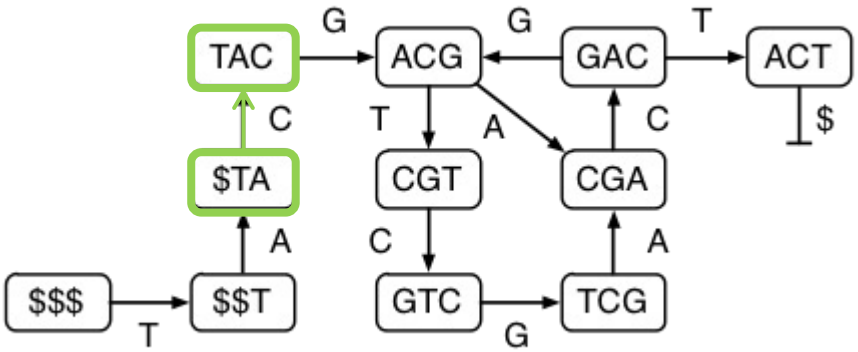
$$select_T(1) = 10$$

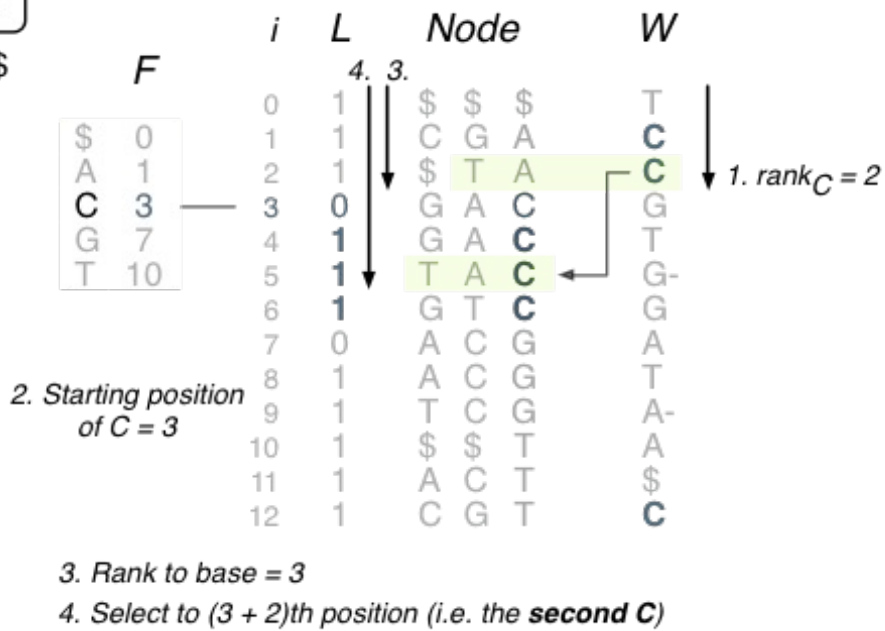
(position of 1'st occurrence of T)

$$select_G(3) = 9$$

(position of 3'rd occurrence of G)

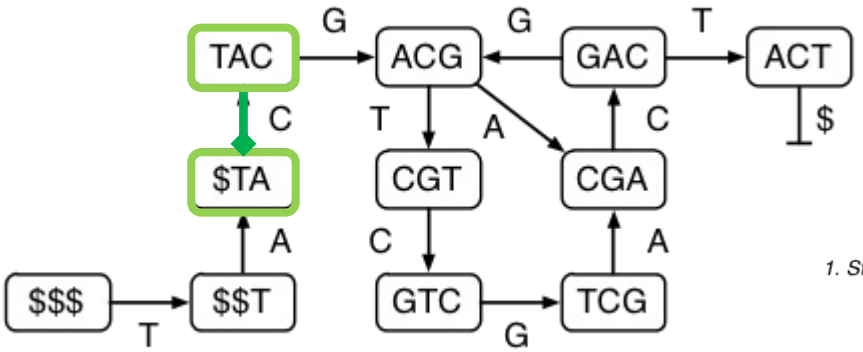
$forward(v, c) = w$

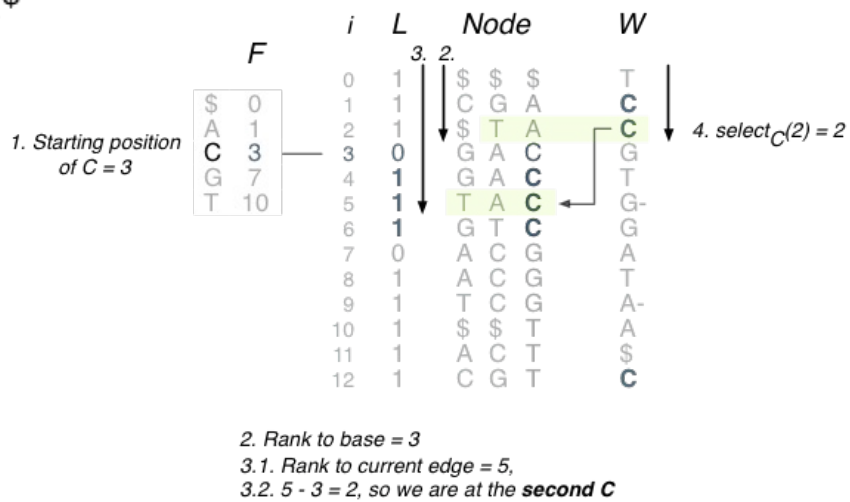




- Find second occurrence of **C** in $Node$ column
- Ignore the first **C** since that has L equal 0
- $O(forward) = 1$ time unit
  1. W access – $O(1)$
  2. rank over W – $O(1)$
  3. rank and select over L – $O(1)$
  4. accessing F – $O(1)$

$backward(w, c) = v \ \ (inverse\_forward)$

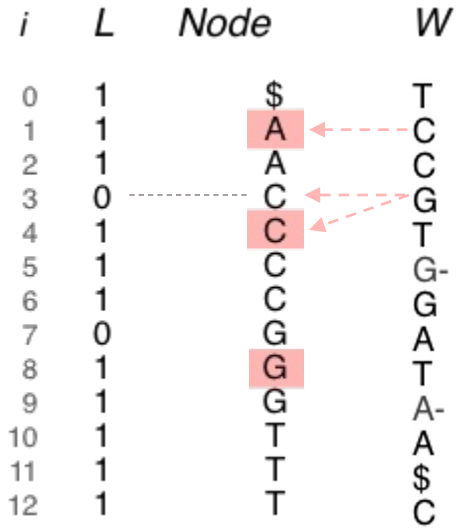




- Find second occurrence of **C** in W column
- We ignore $L[index(C)] == 0$

- $O(backward) = 1$ time unit
  1. Node access – $O(1)$
  2. rank over $Node$ – $O(1)$
  3. rank and select over L – $O(1)$
  4. accessing W – $O(1)$

$label(v) = dna\_sequence$



- What if we need to print out the sequence of Node(e.g. $node\_8$)

- `Execute backward function` $\boldsymbol{k}$ `times`
- Function backward takes $\boldsymbol{\mathcal{O}(1)}$ time
- Repeat the backward function k times ➔

$$\boldsymbol{\mathcal{O}(k)}$$

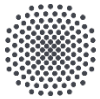
Operations and Time Complexity

Operation	Description	Complexity
$outdegree(v)$	Return number of outgoing edges from node v.	$\mathcal{O}(1)$
$outgoing(v, c)$	From node v, follow the edge labeled by symbol c.	$\mathcal{O}(1)$
$label(v)$	Return (string) label of node v.	$\mathcal{O}(k)$
$indegree(v)$	Return number of incoming edges to node v.	$\mathcal{O}(1)$

Lesson and Conclusion

- Using “creative counting” with two operations $rank$ and $select$ the authors created a memory efficient data structure

- The total space has been reduced to the “closer” to OPT space units
  - i.e.: 300 GB → 2.5 GB
- In principle fairly straight forward to implement
- Tools and libraries(C++ based) like: `MegaHIT`, `BOSS`, `DynaBOSS` and `cosmo` have already implemented the data structure
  - For `MegaHIT` it took 12 hrs to build a graph from 100 GB Metagenomic reads
  - For slightly further future we might use `DynaBOSS`



University of Stuttgart

RNABio²Info

A U C G 1 0 1

Thank you!