Succinct de Bruijn Graphs

Alex Bowe, Kunihiko Sadakane (NII)
Taku Onodera, Tetsuo Shibuya (U.Tokyo)

Twitter @alexbowe

Blog alexbowe.com/succinct-debruijn-graphs

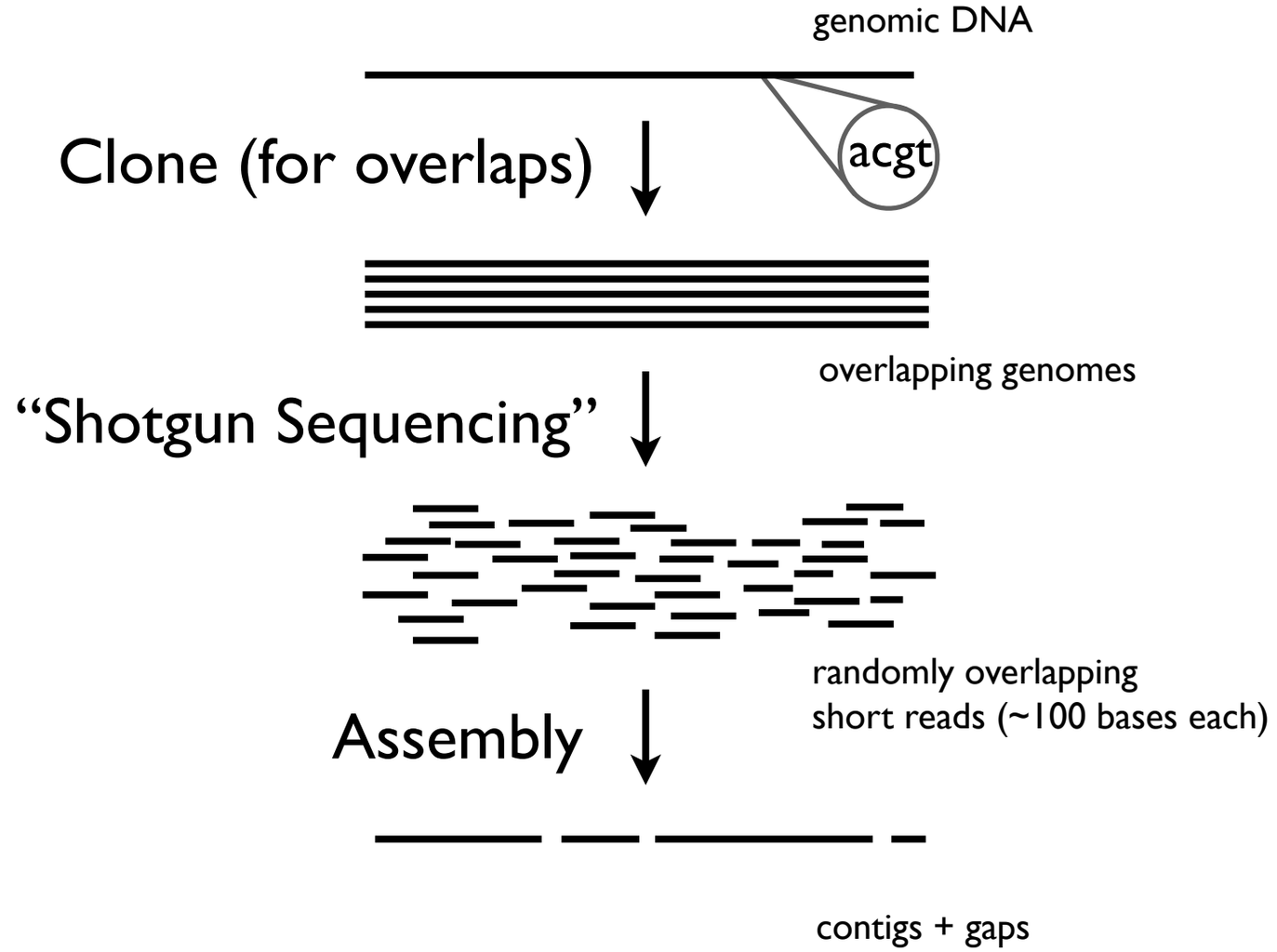
Demo Code
(~100 lines of Python) github.com/alexbowe/debby

Overview

- DNA assembly background
- Comparison of current assemblers
- Our representation and operations
- Conclusion

DNA Sequencing

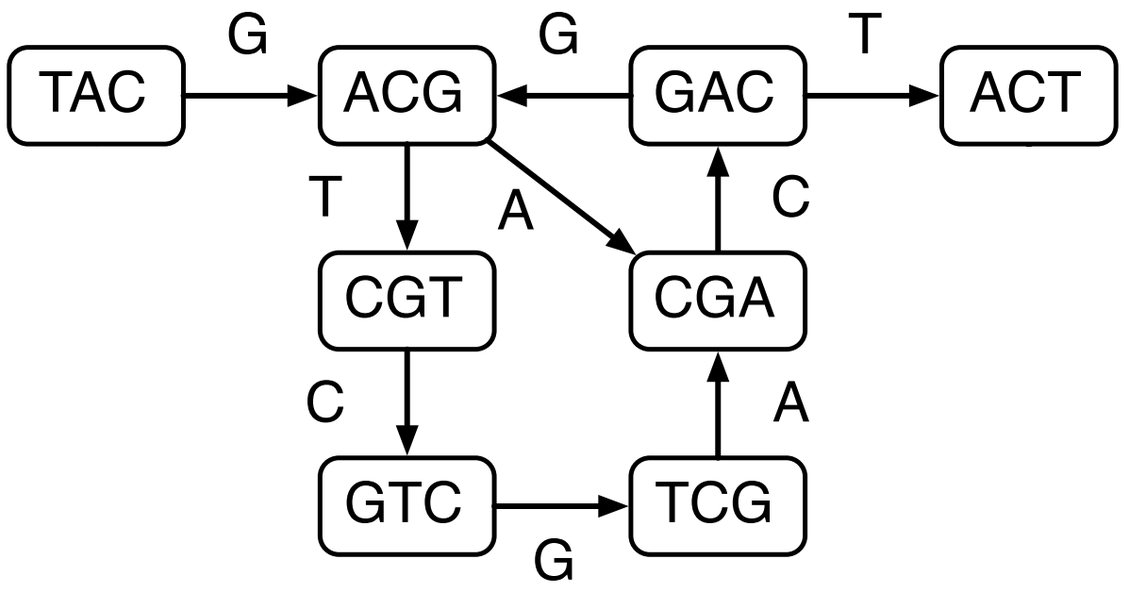
- Objective: Read genome into computer *(~3.2 billion bases)*
- Problem: molecule too small to read entirely
- Break it into **random overlapping "short reads"**
- Algorithms required to **assemble** these back in the (close to) correct order



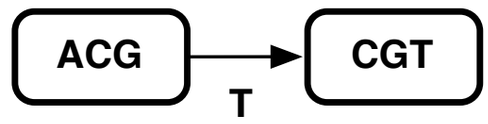
de Bruijn Graphs

T = TACGACGTCGACT *sequencing read*

↓ *extract overlapping kmers (fixed k-length substrings)*



ACG
CGT



edge if k-1 overlap

assembly methods differ, but proposed as finding Eulerian tour

Scaling

Assembler	**Approach**	**Bits** *per edge*
ABySS [Simpson et al. 2009]	Distributed hash table	~300
Gossamer [Conway et al. 2012]	Succinct bit vector	28.5
[Pell et al. 2011]	Bloom filter with false pos. edges	4~9
Minia [Cikhi, Rizk 2012]	Bloom filter with aux. struct	13.5

Depending on k (here k = 27)

Ours: 4 + o(1) bits per edge for m edges (independent of k)
(~3 bits per edge after compression)

Preliminary Data

human genome (NA18507)

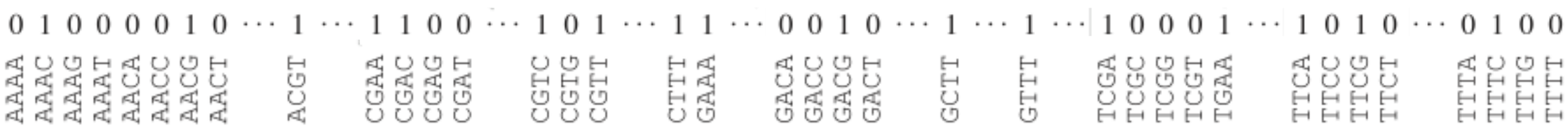
Assembler	**Gigabytes** ***total***	**Time** ***total***
ABySS [Simpson et al. 2009]	336 GB	**15 hours**
Gossamer [Conway et al. 2012]	32 GB	50 hours
Minia [Cikhi, Rizk 2012]	5.7 GB	23 hours 6.4 hours for assembly
Ours	**2.5 GB**	120 hours 4.5 hours for assembly

k = 27

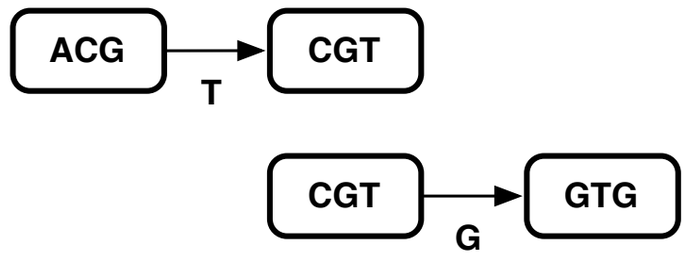
m = 5.3 billion

While Minia takes around 23 hours, ours takes 120 hours because we use an unoptimised k-mer counting method (disk sort).

Gossamer



- Enumerate all possible (k+1)-mers
  k-mer is node label, 1-mer is edge label + target node



- Implicitly store in succinct *sparse* bit vector
- Redundancy: edges have overlap - representing each node multiple times
- We use this overlapping property for compression

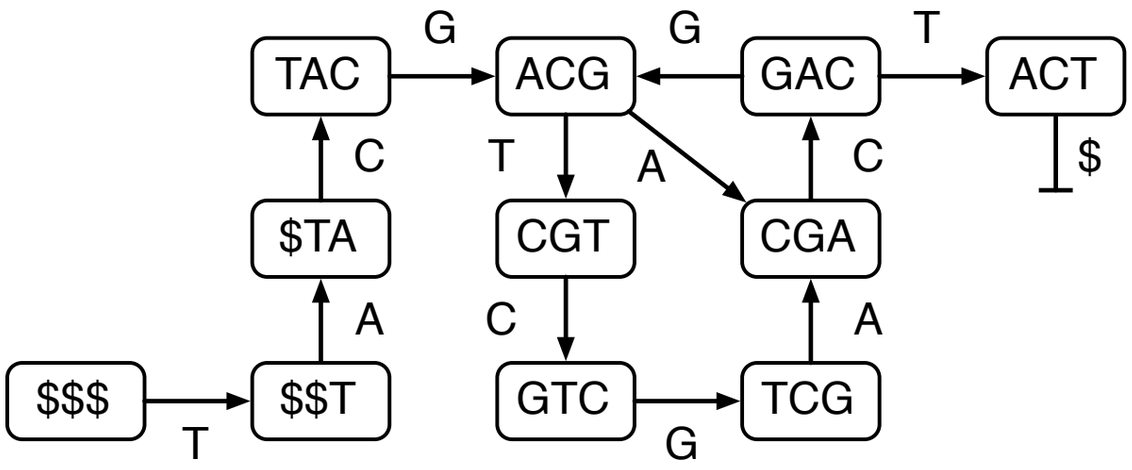
Minia

- Store k-mers in bloom filter
- Store critical false positives in aux. struct.
- Start at a random node:
  - generate each possible outgoing edge
  - test if it exists in the bloom filter
  - double check using aux. struct.

Our Implementation

Sort <node label, edge label> pairs in "colex" order of node label *(can easily be distributed or done on disk)*

T = $$$TACGACGTCGACT$

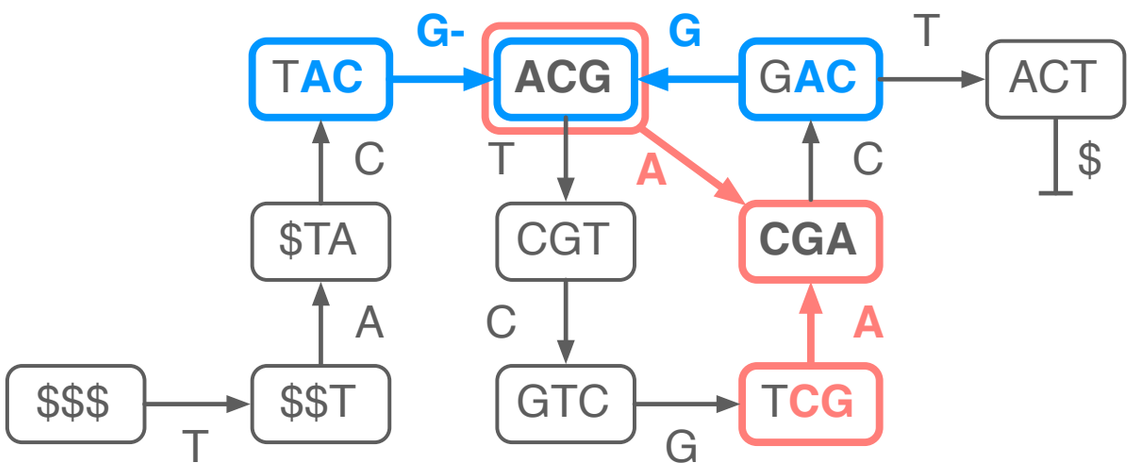


$ signs - dummy edges (every node needs at least 1 incoming and 1 outgoing edge)

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

Our Implementation

- W has minus flag when we have witnessed the same edge label for the same suffix
- This differentiates incoming edges to same node



Note that W[i] ∈ A ∪ A⁻
(A is alphabet)

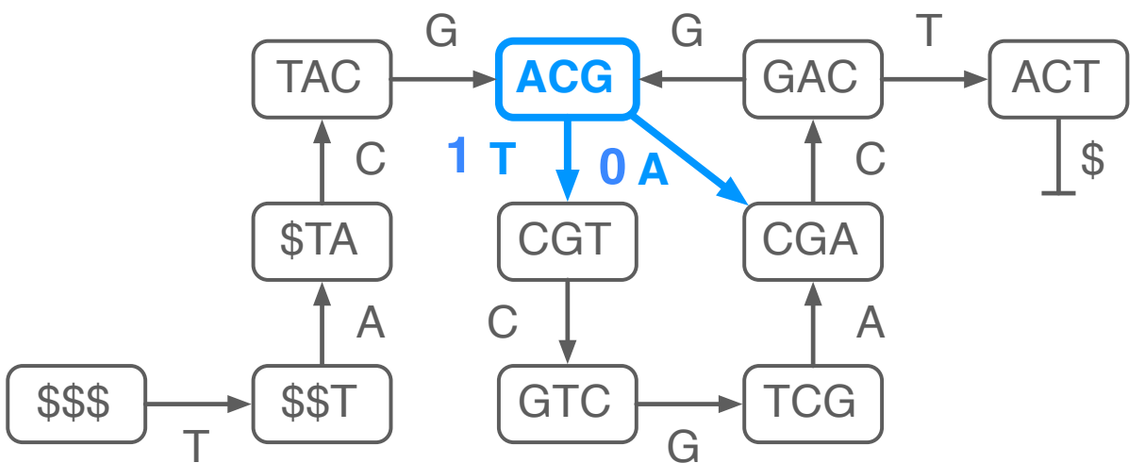
Node	*W*
$ $ $	T
C G A	C
$ T A	C
G A C	G
G A C	T
T A C	G-
G T C	G
A C G	A
A C G	T
T C G	A-
$ $ T	A
A C T	$
C G T	C

Our Implementation

L (last) bit vector: 0 if not the last edge of node

this indicates the ranges of identical node labels: 0*1

1:1 correspondence of nodes with last[i] = 1 and $W \in A$

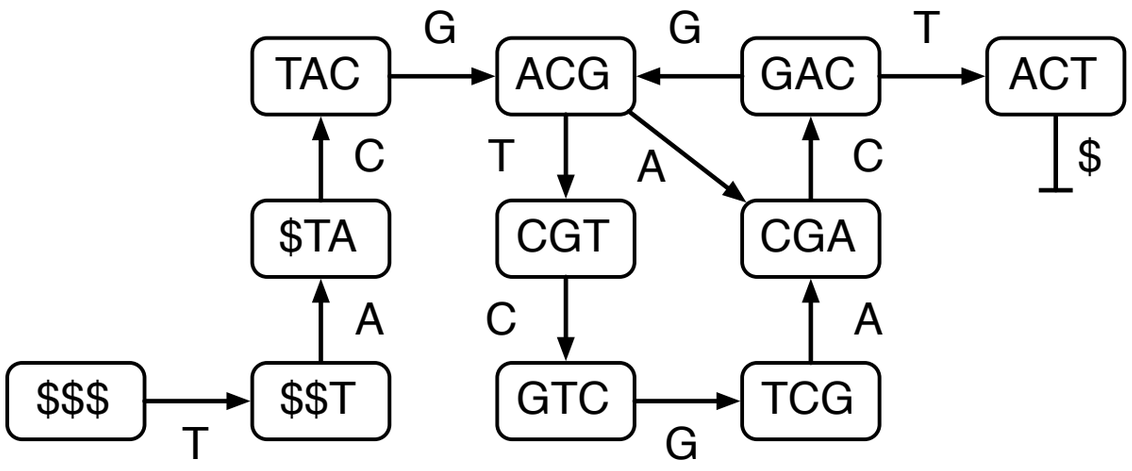


L	*Node*			*W*
1	$	$	$	T
1	C	G	A	C
1	$	T	A	C
0	G	A	C	G
1	G	A	C	T
1	T	A	C	G-
1	G	T	C	G
0	A	C	G	A
1	A	C	G	T
1	T	C	G	A-
1	$	$	T	A
1	A	C	T	$
1	C	G	T	C

Our Implementation

This creates two ways to think about indexing:

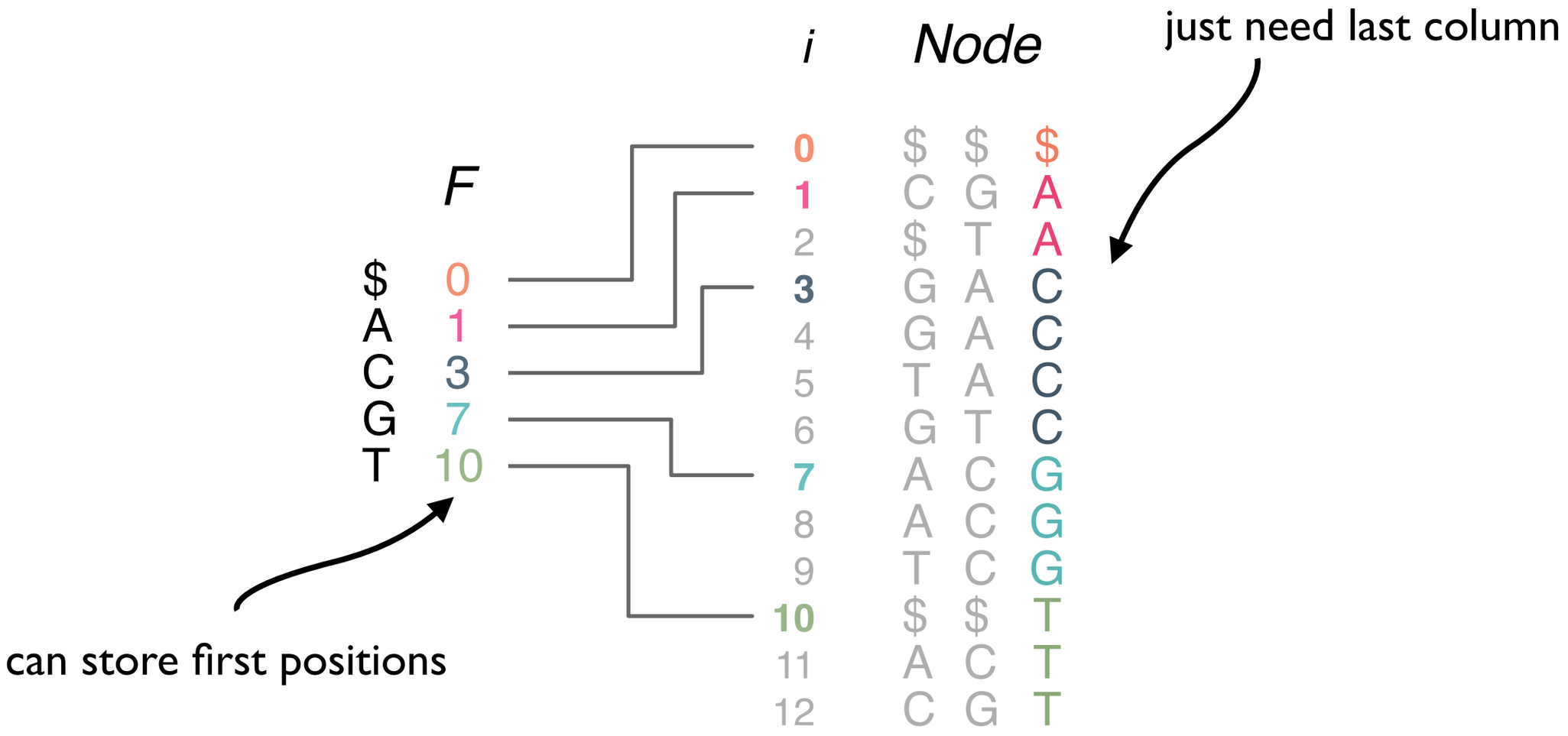
- By vertex (v)
- By edge/row (i)



v	*i*	*L*	*Node*			*W*
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	T
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	**7**	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

Our Implementation

Don’t need to store Node labels



In Total

for m edges:

	L	*W*
	1	T
	1	C
F	1	C
	0	G
0	1	T
1	1	G-
3	1	G
7	0	A
10	1	T
	1	A-
	1	A
	1	$
	1	C

σ log(m) bits

m bits

m log(2σ) = m + m log(σ) bits

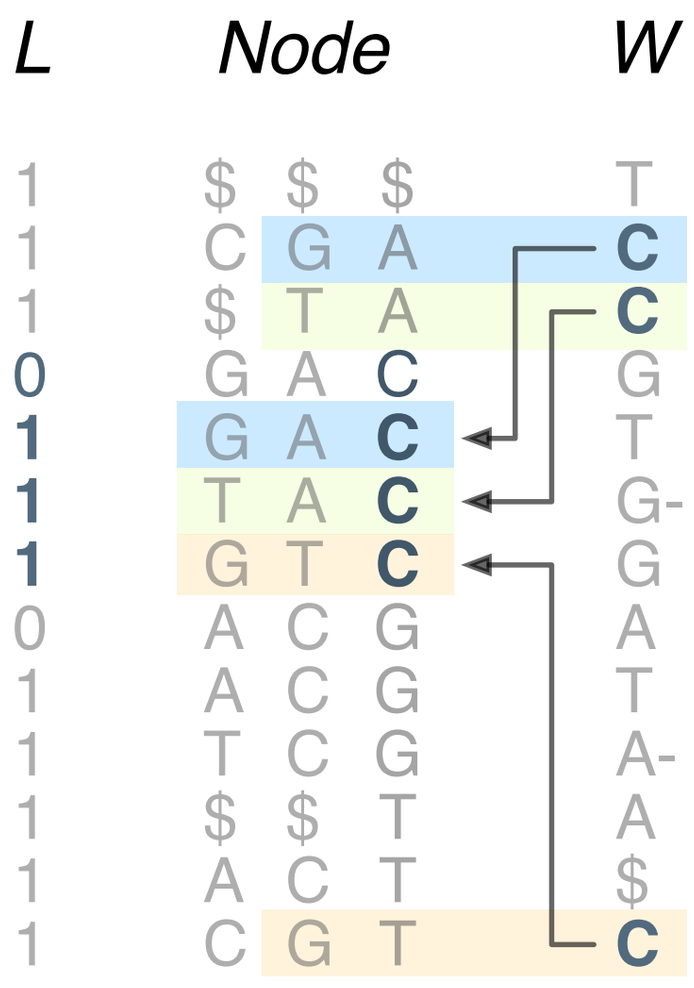
2 + log(σ) + o(1) bits per edge

How Do We Navigate?

- W maintains the relative ordering of each symbol occurrence

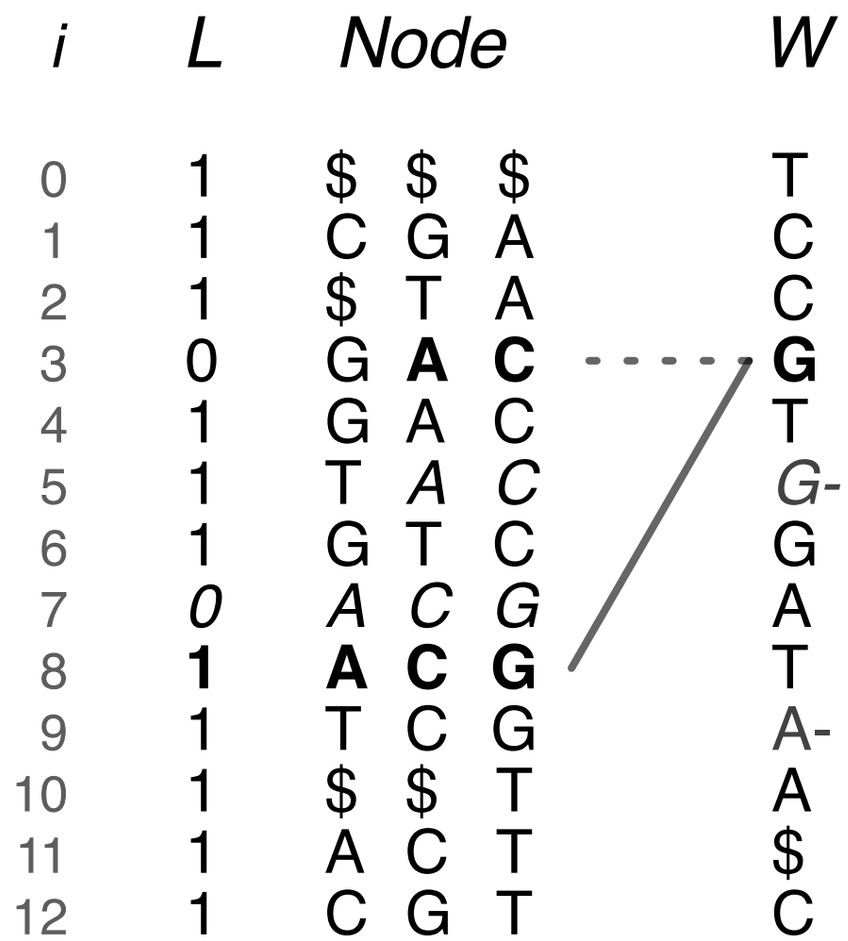
- To follow in either direction, we can count the number of previous occurrences (ignoring if L[i]=0 or minus-flagged)

- Counting and finding the *ith* occurence can be performed in O(1) time using a rank/select index

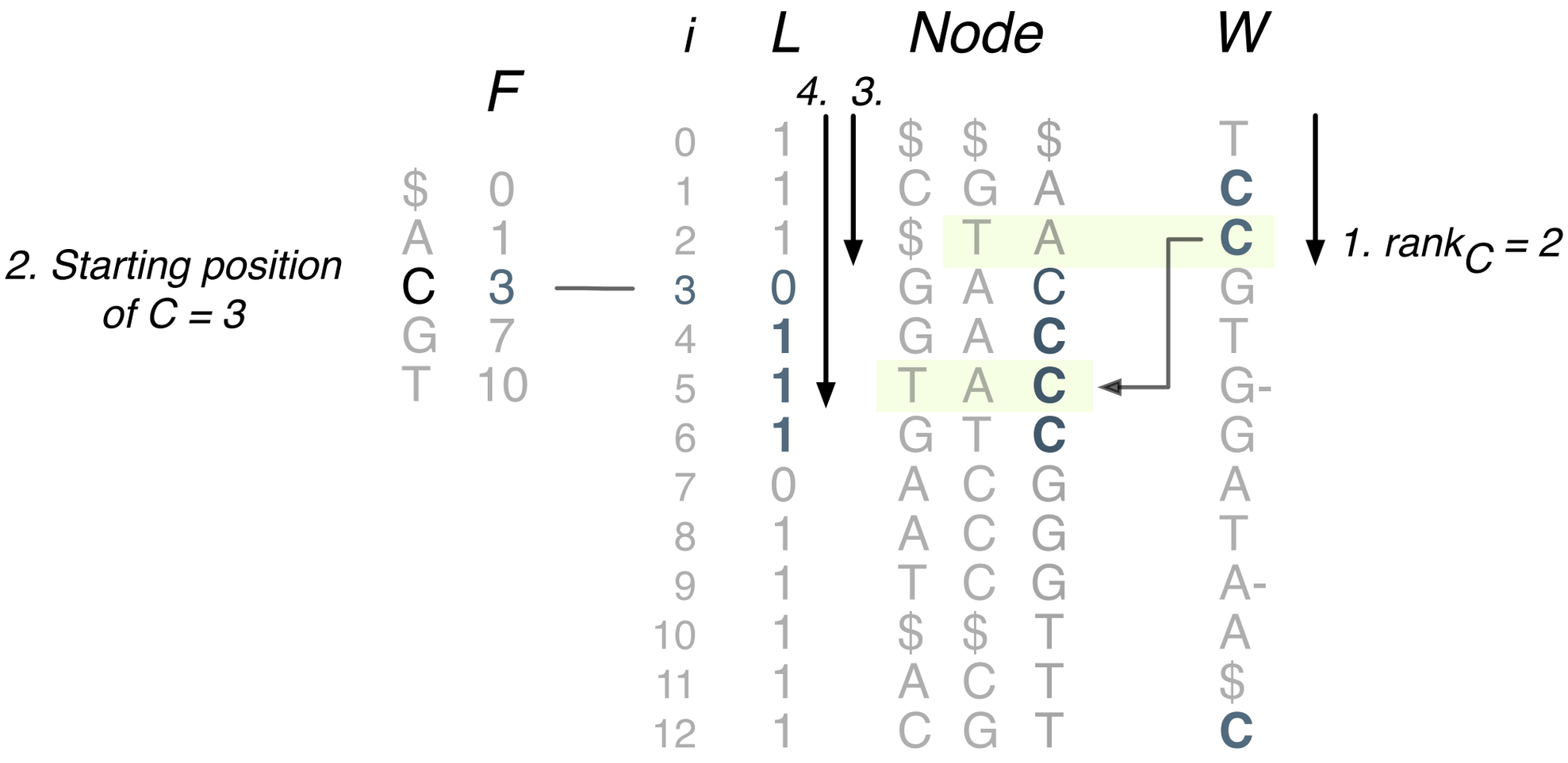


fwd() & bwd()

Function	Description	Complexity
fwd(i)	Return index of *last* (L = 1) edge of node pointed to by edge i	O(1)
bwd(j)	Return index of *first* (no minus) edge that points to the node that edge j leaves	O(1)



Example: fwd(2)



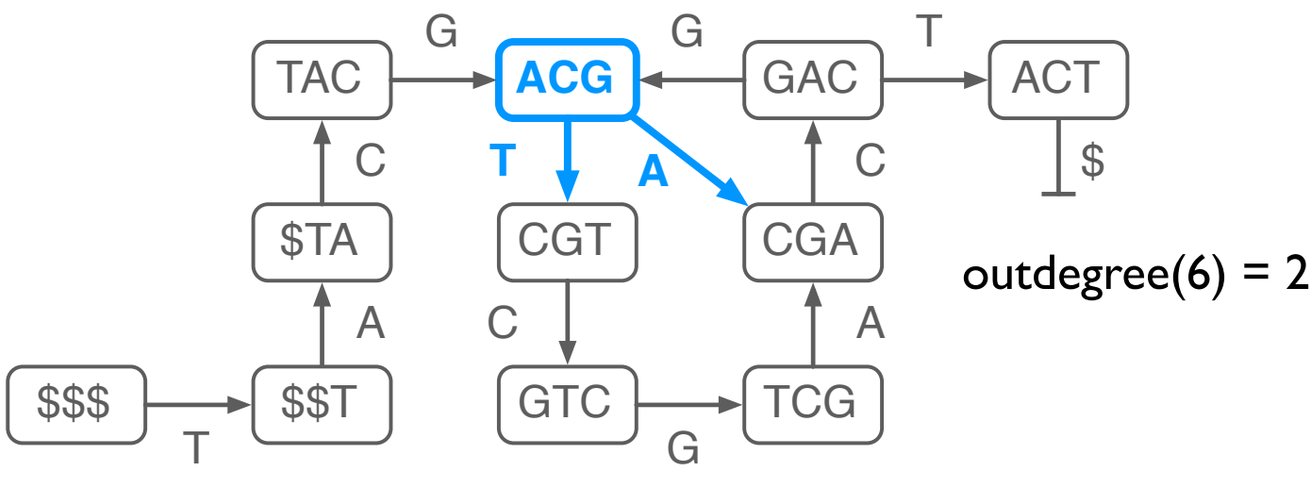
Interface

Function	**Description**	**Complexity**
outdegree(v)	return # outgoing edges	O(1)
outgoing(v,c)	follow edge c from v	O(1)
node(v)	return label of v	O(k)
indegree(v)	return # incoming edges	O(1)
incoming(v, c)	return pred. of v starting with c	O(k log σ)

outgoing edges - O(1)

outdegree(v)

- All outgoing edges are contiguous, last: 0*1
- Count zeros, add one...
- $select_1(L, v) - select_1(L, v-1)$



outdegree(6) = 2

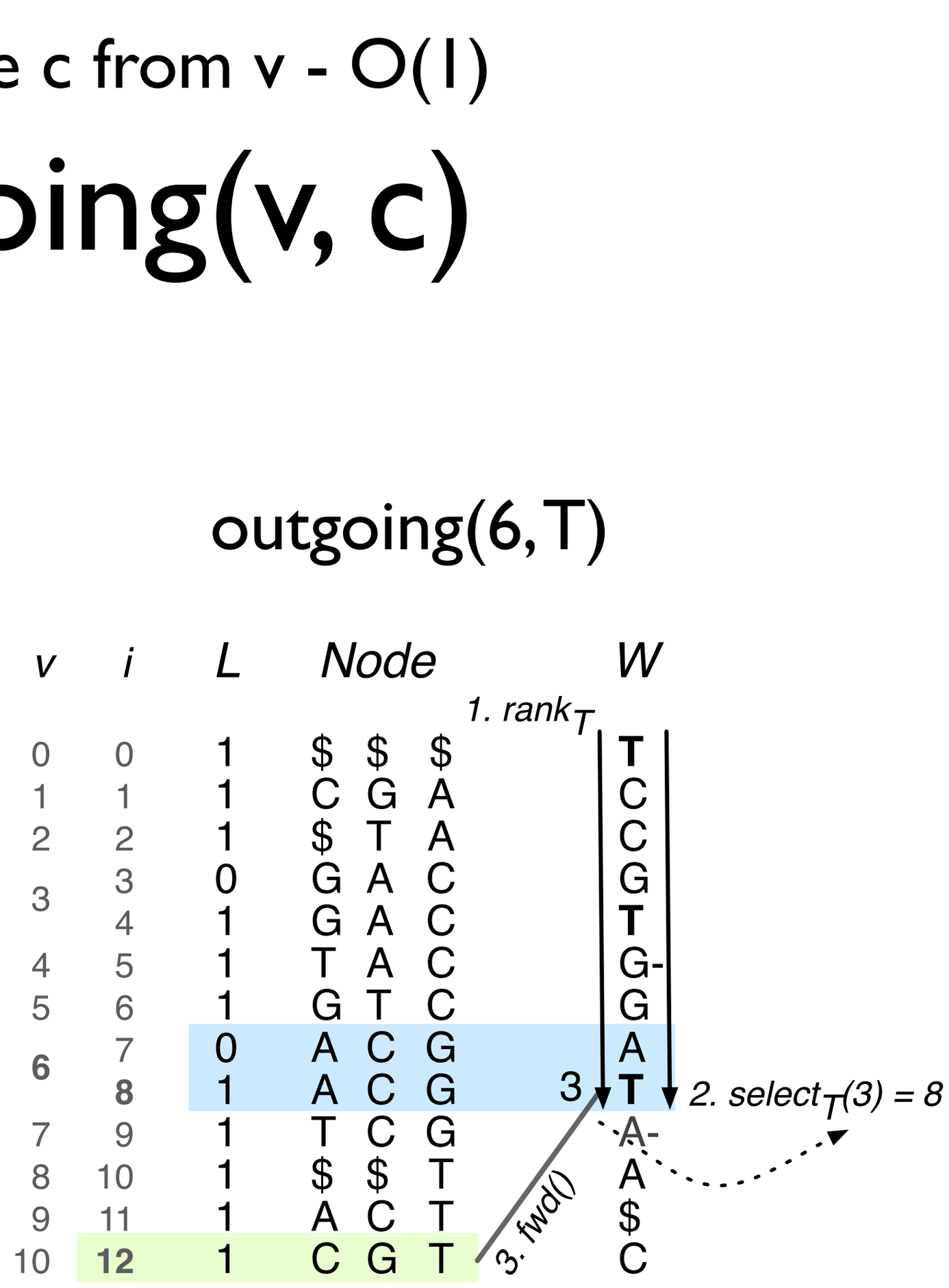
v	*i*	*L*	*Node*			*W*
						select(6) - select(5)
0	0	**1**	$	$	$	T
1	1	**1**	C	G	A	C
2	2	**1**	$	T	A	C
3	3	**0**	G	A	C	G
	4	**1**	G	A	C	T
4	5	**1**	T	A	C	G-
5	**6**	**1**	G	T	C	G
6	7	**0**	A	C	G	A
	8	**1**	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

follow edge c from v - O(1)

outgoing(v, c)

- Search outgoing edges for c or c^-
- select(rank(i)): position of last occ. in the range [0,i]
- $x = select_c(W, rank_c(W, v))$
- if not in outgoing-edge range (known from outdegree) try c^-
- return fwd(x)

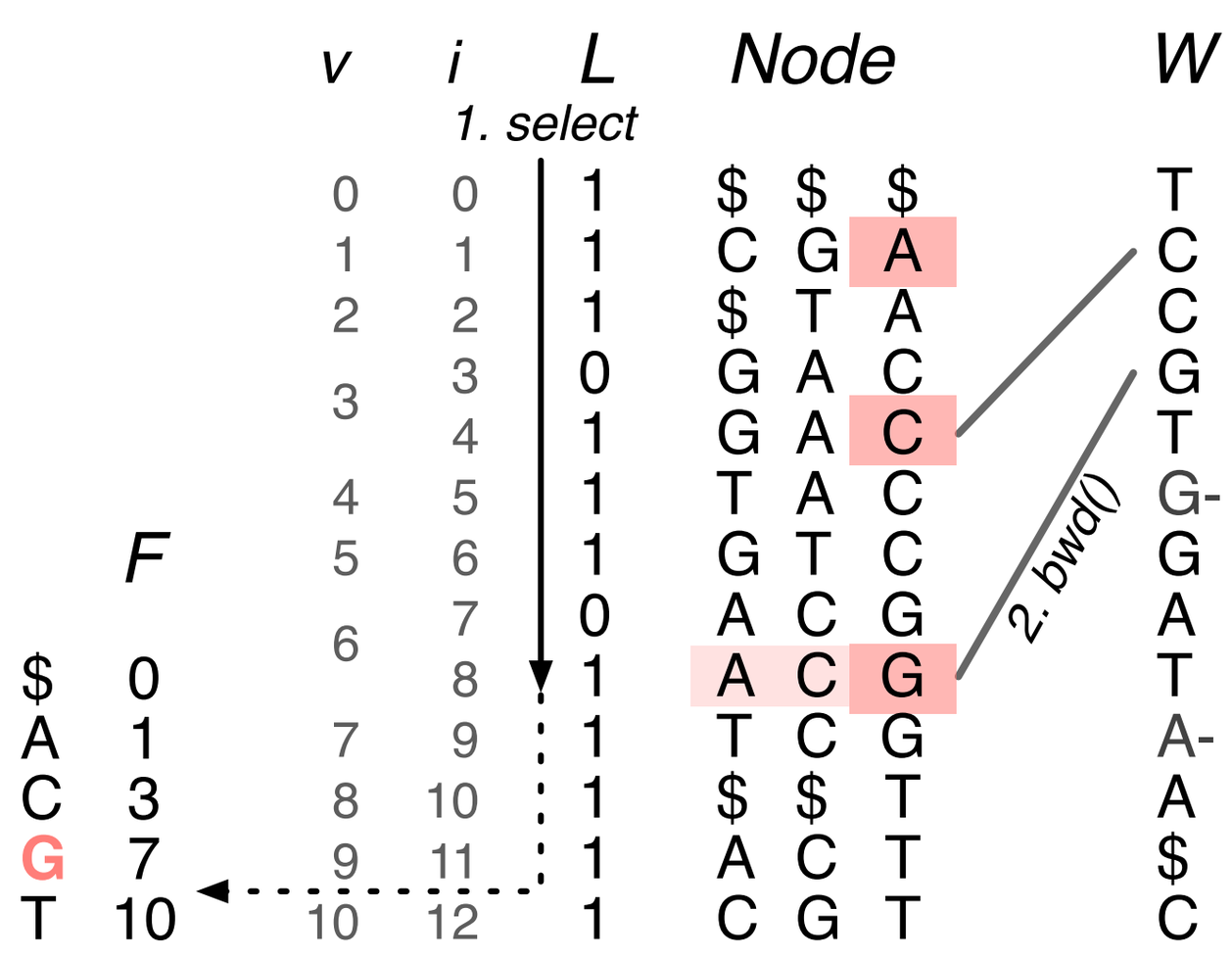
outgoing(6, T)



return label of v - O(k)

node(v)

- Reminder: Node[] not stored, but last (kth) letter is stored in F.
- Calculate bwd() k times, using resulting indexes to reverse lookup in F.



Summary

- Compact representation of de Bruijn graph
  - m(2 + log σ + o(1)) bits
  - $4m$ + o(m) bits for DNA,
    ~3 bits per edge after compression
  - navigation operations done quickly
- alexbowe.com/succinct-debruijn-graphs

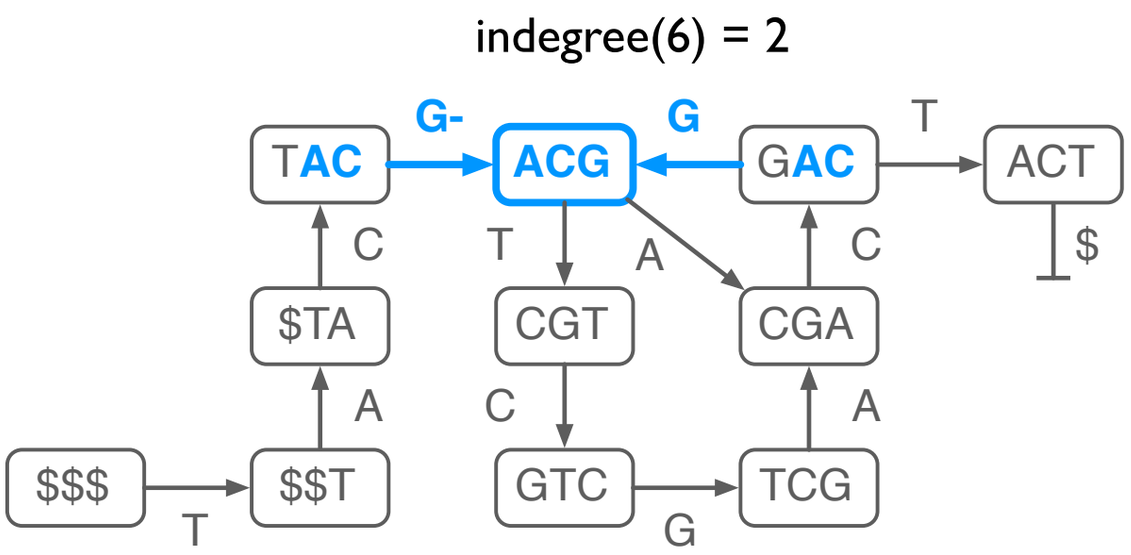
Thank you (:

Questions?

incoming edges - O(1)

indegree(v)

- use bwd() to give us *first* predecessor node
- if there are more, they sort after as c-
- rank/select to next c, subtract ranks of c- to find degree



v	*i*	*L*	*Node*			*W*
0	0	1	$	$	$	T
1	1	1	C	G	A	C
2	2	1	$	T	A	C
3	3	0	G	A	C	G
	4	1	G	A	C	T
4	5	1	T	A	C	G-
5	6	1	G	T	C	G
6	7	0	A	C	G	A
	8	1	A	C	G	T
7	9	1	T	C	G	A-
8	10	1	$	$	T	A
9	11	1	A	C	T	$
10	12	1	C	G	T	C

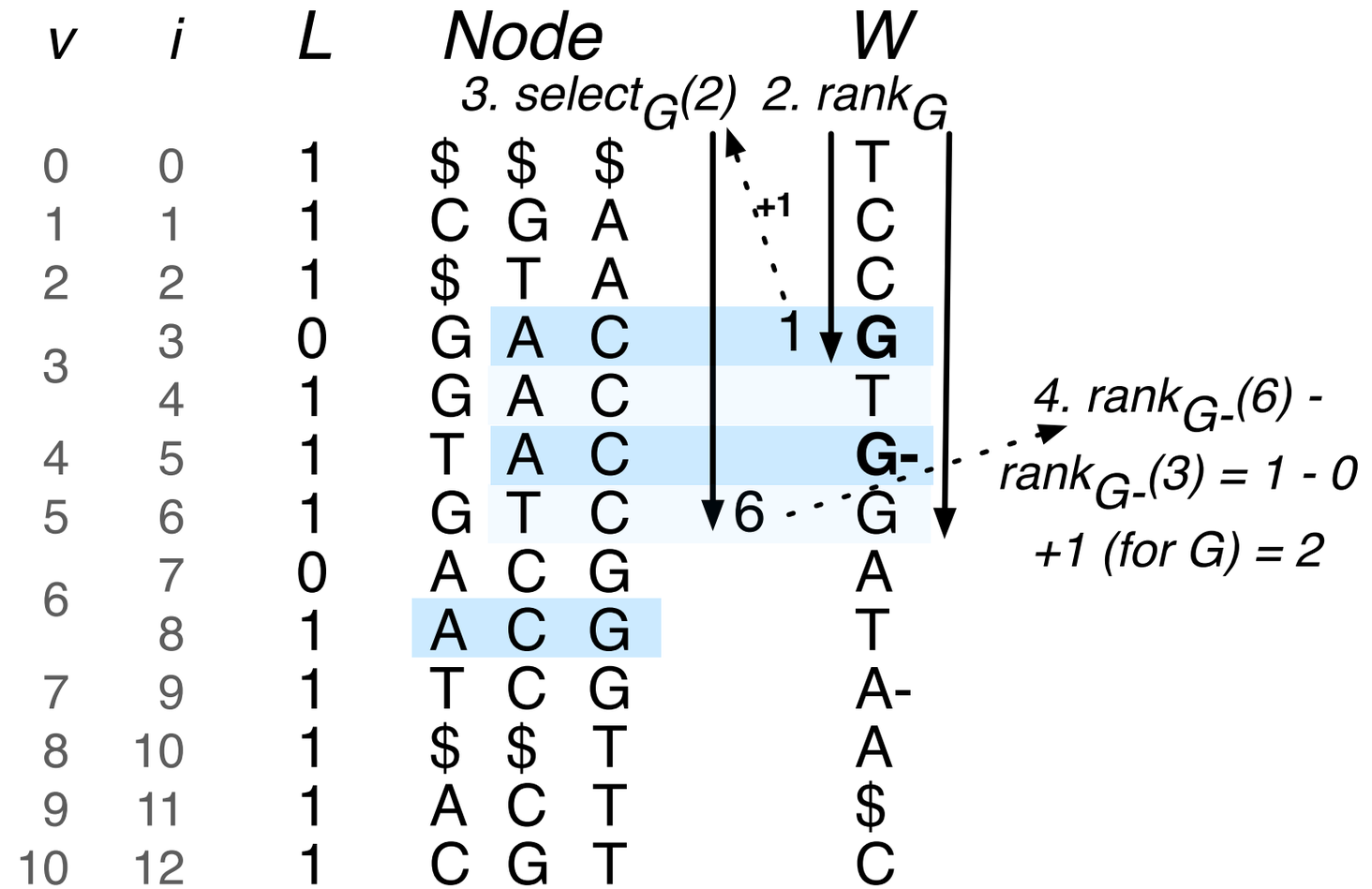
1. bwd()

predecessors

incoming edges - O(1)

indegree(v)

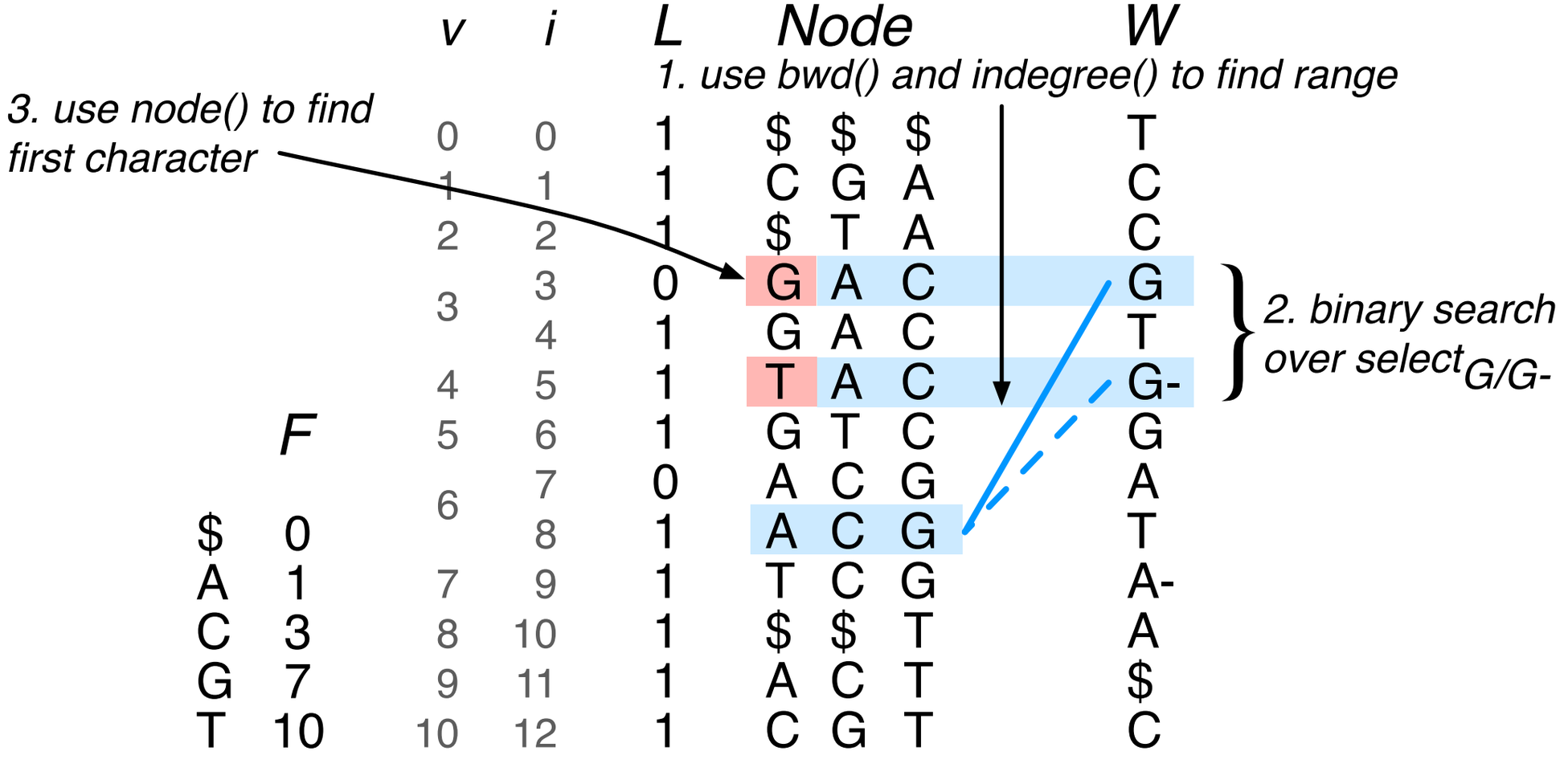
- x = rank to count previous c edges
- select(x+1) finds *next* c edge
- rank c- to count occurrences between



predecessor of v starting with c - O(k log σ)

incoming(v, c)

- Similar to indegree() to locate predecessors
- These nodes differ only in their first character, and are sorted
- Can use node() to find first character - O(k)
- binary search: O(log σ) time



Thank you (:

Okay, really finished this time.

Construction

- Batch
  - Sort all (k+1)-mers, remove duplicates
- Online
  - given graph for string of length N-1
  - modify it for string of length N made by appended characters
  - O(Nk log m/log log m) time, O(Nk) time if σ is small
- Iterative
  - Construct k-dim. graph from (k-1)-dim. graph
  - O(Nk) time, O(*m*σ) bits space