De Bruijn Graph Assembly
Ben Langmead

Johns Hopkins
Whiting School of Engineering
Department of Computer Science

Please sign guestbook (www.langmead-lab.org/teaching-materials) to tell me briefly how you are using the slides. For original Keynote files, email me (ben.langmead@gmail.com).
Different kind of graph

“tomorrow and tomorrow and tomorrow”

An edge represents an ordered pair of adjacent words in the input

Multigraph: there can be more than one edge from node A to node B
De Bruijn graph

L/R 2-mers: AA, AA, AA, AB, AB, BB, BB, BB, BB, BB, BB, BB, BA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

One edge per k-mer
One node per distinct k-1-mer

genome: AAABBBBA
Walk crossing each edge exactly once gives a reconstruction of the genome
De Bruijn graph

Walk crossing each edge exactly once gives a reconstruction of the genome. This is an *Eulerian walk*. 

AAABBBBBBA
De Bruijn graph

Aside: how do you pronounce "De Bruijn"?

There is debate:

https://www.biostars.org/p/7186/

I still don't quite know. I say "De Broin" (rhymes with "groin")

I asked a Dutch person once; his pronunciation sounded more like "De Brown"

Nicolaas Govert
de Bruijn
1918 -- 2012
Directed multigraph

Directed **multigraph** $G(V, E)$ consists of set of vertices, $V$ and **multiset** of directed edges, $E$

Otherwise, like a directed graph

Node’s *indegree* = # incoming edges

Node’s *outdegree* = # outgoing edges

De Bruijn graph is a directed multigraph

$$V = \{a, b, c, d\}$$

$$E = \{(a, b), (a, b), (a, b), (a, c), (c, b)\}$$
Eulerian walk definitions and statements

Node is *balanced* if indegree equals outdegree

Node is *semi-balanced* if indegree differs from outdegree by 1

Graph is *connected* if each node can be reached by some other node

_Eulerian walk_ visits each edge exactly once

Not all graphs have Eulerian walks. Graphs that do are *Eulerian*. (For simplicity, we won’t distinguish Eulerian from semi-Eulerian.)

A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced

Jones and Pevzner section 8.8
De Bruijn graph

Back to de Bruijn graph

Is it Eulerian? Yes

Argument 1: $AA \rightarrow AA \rightarrow AB \rightarrow BB \rightarrow BB \rightarrow BA$

Argument 2: $AA$ and $BA$ are semi-balanced, $AB$ and $BB$ are balanced
De Bruijn graph

A procedure for making a de Bruijn graph for a genome

Assume “perfect sequencing”: each genome $k$-mer is sequenced exactly once with no errors

Pick a substring length $k$: 5

Start with an input string: `a_long_long_long_time`

Take each $k$ mer and split into left and right $k-1$ mers

Add $k-1$ mers as nodes to de Bruijn graph (if not already there), add edge from left $k-1$ mer to right $k-1$ mer
De Bruijn graph

First 8 $k$-mer additions, $k = 5$

a_long_long_long_time
De Bruijn graph

Last 5 k-mer additions, $k = 5$

a_long_long_long_time

Finished graph
De Bruijn graph

Procedure yields Eulerian graph. Why?

Node for $k$-1-mer from left end is semi-balanced with one more outgoing edge than incoming *

Node for $k$-1-mer at right end is semi-balanced with one more incoming than outgoing *

Other nodes are balanced since # times $k$-1-mer occurs as a left $k$-1-mer = # times it occurs as a right $k$-1-mer

* Unless left- and right-most $k$-1-mers are equal
De Bruijn graph

What string does the Eulerian path spell out?

\texttt{a_long_long_long_time}

The original string! No collapsing!
De Bruijn graph builder implementation

class DeBruijnGraph:
    """ A de Bruijn multigraph built from a collection of strings. User supplies strings and k-mer length k. Nodes of the de Bruijn graph are k-1-mers and edges join a left k-1-mer to a right k-1-mer. """

    @staticmethod
def chop(st, k):
        """ Chop a string up into k mers of given length """
        for i in xrange(0, len(st)-(k-1)): yield st[i:i+k]

class Node:
    """ Node in a de Bruijn graph, representing a k-1 mer """
def __init__(self, km1mer):
    self.km1mer = km1mer

def __hash__(self):
    return hash(self.km1mer)

def __init__(self, strIter, k):
    """ Build de Bruijn multigraph given strings and k-mer length k """
    self.G = {} # multimap from nodes to neighbors
    self.nodes = {} # maps k-1-mers to Node objects
    self.k = k
    for st in strIter:
        for kmer in self.chop(st, k):
            km1L, km1R = kmer[:-1], kmer[1:]
            nodeL, nodeR = None, None
            if km1L in self.nodes:
                nodeL = self.nodes[km1L]
            else:
                nodeL = self.Node(km1L)
            if km1R in self.nodes:
                nodeR = self.nodes[km1R]
            else:
                nodeR = self.Node(km1R)
            self.G.setdefault(nodeL, []).append(nodeR)
De Bruijn graph

For Eulerian graph, Eulerian walk can be found in $O(|E|)$ time. $|E|$ is # edges.

Convert graph into one with Eulerian cycle (add an edge to make all nodes balanced), then use this recursive procedure

```
# Make all nodes balanced, if not already

tour = []
# Pick arbitrary node
src = g.iterkeys().next()

def __visit(n):
    while len(g[n]) > 0:
        dst = g[n].pop()
        __visit(dst)
        tour.append(dst)
__visit(src)
# Reverse order, omit repeated node
tour = tour[::-1][:-1]
# Turn tour into walk, if necessary
```

Insight: If C is a cycle in an Eulerian graph, then after removing edges of C, remaining connected components are also Eulerian
De Bruijn graph

Full illustrative de Bruijn graph and Eulerian walk implementation:


Example where Eulerian walk gives correct answer for small $k$ whereas Greedy-SCS could spuriously collapse repeat:

```python
>>> G = DeBruijnGraph(["a_long_long_long_time"], 5)
>>> print G.eulerianWalkOrCycle()
['a_lo', '_lon', 'long', 'ong_', 'ng_l', 'g_lo',
'_lon', 'long', 'ong_', 'ng_l', 'g_lo', '_lon',
'long', 'ong_', 'ng_t', 'g_ti', '_tim', 'time']
```
De Bruijn graph

```python
>>> st = "to_everyThing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()  # Fast! Linear in # edges
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
"to_every_thing_turn_turn_turn_there_is_a_season"


Recall: This is not generally possible or tractable in the overlap/SCS formulation
De Bruijn graph

Assuming perfect sequencing, procedure yields graph with Eulerian walk that can be found efficiently.

We saw cases where Eulerian walk corresponds to the original superstring. Is this always the case?
De Bruijn graph

Problem 1: Repeats still cause misassemblies

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY
ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

Problem 2:

We’ve been building DBGs assuming “perfect” sequencing: each $k$-mer reported exactly once, no mistakes. Real datasets aren’t like that.
Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome.

Collapsing: 

```
a_long_long_long_time
```

Shuffling:

```
``
De Bruijn graph

- Genome: **AAABBBBA**
- 3-mers: **AAA, AAB, ABB, BBB, BBB, BBA**
- L/R 2-mers: **AA, AA, AA, AB, AB, BB, BB, BB, BB, BB, BB, BA**

**Notes:**
- One edge per $k$-mer
- One node per distinct $k-1$-mer
De Bruijn graph

```python
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()  # Fast! Linear in # edges
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
"to_every_thing_turn_turn_turn_there_is_a_season"
```

De Bruijn graph

Case where $k = 4$ works:

```python
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

But $k = 3$ does not:

```python
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
```
De Bruijn graph

Case where $k = 4$ works:

```python
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
    to_every_thing_turn_turn_turn_there_is_a_season
```

But $k = 3$ does not:

```python
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
    to_every_turn_turn_thing_turn_there_is_a_season
```

Due to repeats that are unresolvable at $k = 3$
De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY
ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

Problem 2:
We’ve been building DBGs assuming “perfect” sequencing: each $k$-mer reported exactly once, no mistakes. Real datasets aren’t like that.
De Bruijn graph

Gaps in coverage (missing $k$-mers) lead to disconnected or non-Eulerian graph

Graph for a_long_long_long_time, $k = 5$: 
De Bruijn graph

Gaps in coverage (missing \( k \)-mers) lead to disconnected or non-Eulerian graph

Graph for \texttt{a\_long\_long\_long\_time}, \( k = 5 \) but omitting \texttt{ong\_t}:

\texttt{ng\_l} \begin{array}{c}
\rightarrow \texttt{g\_lo} \\
\rightarrow \texttt{a\_lo} \\
\rightarrow \texttt{\_lon} \\
\rightarrow \texttt{long} \\
\rightarrow \texttt{ong\_}
\end{array}

\texttt{ng\_t} \begin{array}{c}
\rightarrow \texttt{g\_ti} \\
\rightarrow \texttt{\_tim} \\
\rightarrow \texttt{time}
\end{array}
De Bruijn graph

Coverage *differences* make graph non-Eulerian

Graph for *a_long_long_long_time*,
\( k = 5 \), with *extra copy* of *ong_t*:

4 *semi-balanced* nodes
De Bruijn graph

Errors and differences between chromosomes also lead to non-Eulerian graphs

Graph for `a_long_long_long_time`, \( k = 5 \) but with error that turns one copy of `long_` into `lxng_`
De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical

Uneven coverage, sequencing errors, etc make graph non-Eulerian

Even if graph were Eulerian, repeats yield many possible walks


*De Bruijn Superwalk Problem* (DBSP) seeks a walk over the De Bruijn graph, where walk contains each read as a *subwalk*

Proven NP-hard!