GRAPHS AND GENOME ASSEMBLIES

-Why the de Bruijn graph?

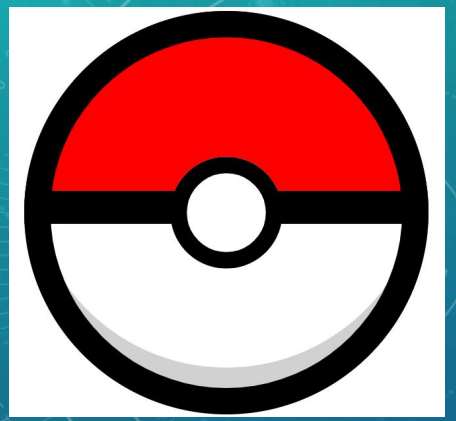
FOR THE IOB CLUB OF ALGORITHM @UGA

CONTRIBUTORS:

OUTLINE

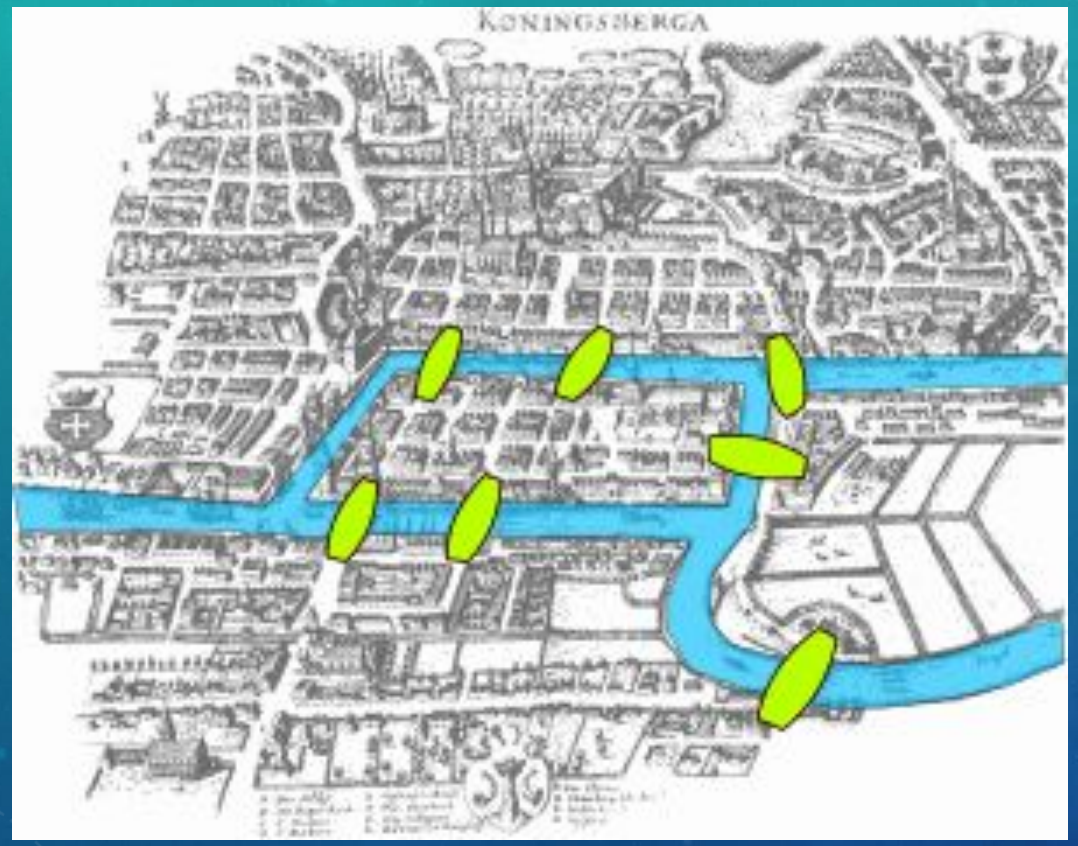
- Basic concepts in graph theory
- Brief Introduction of Euler cycles, Hamilton cycles
- ~~Brief discussion on the NP-COMPLETE~~
- Brief Introduction of the superstring sequence problem
- de Bruijn's solution to the superstring sequence problem
- Why is de Bruijn graph related to the genome assembly?

COULD YOU DRAW THE POKEBALL?



Challenge: What about without lifting your pen from the paper and without repeating the parts already drawn?

SEVEN BRIDGES OF KÖNIGSBERG

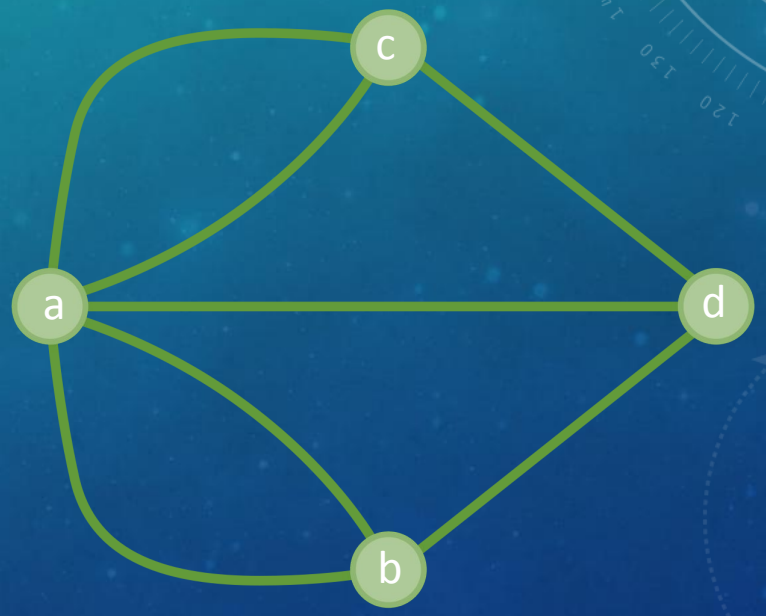


Nailed it!

Leonhard Euler 1707-1783
Genius

THE ORIGIN OF NETWORK GRAPHS

- V: **V**ertex/Vertices (node/nodes) #SET
- E: **E**dge #SET
- G=(V,E)
- **An Eulerian trail (or Eulerian path) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices)**
- **Eulerian cycle: a path through the graph that visits every edge exactly once and returns back where it started**

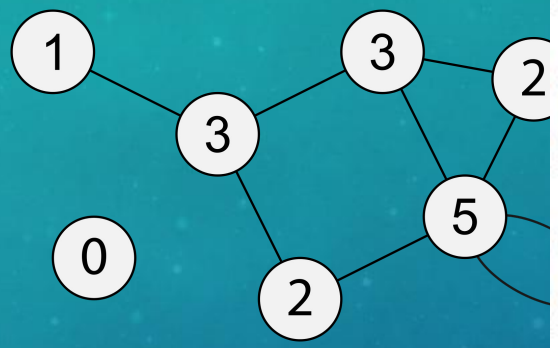


THEOREM

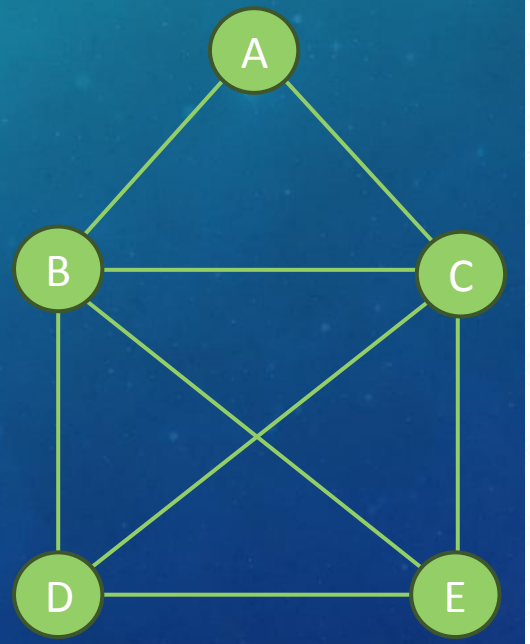
- An Eulerian cycle exists if and only if the degrees of all vertices are even.
- And an Eulerian path exists if and only if the number of vertices with odd degrees is two (or zero, in the case of the existence of a Eulerian cycle).
- In addition, of course, the graph must be sufficiently connected (i.e., if you remove all isolated vertices from it, you should get a connected graph).

EULER'S PATH & CYCLE

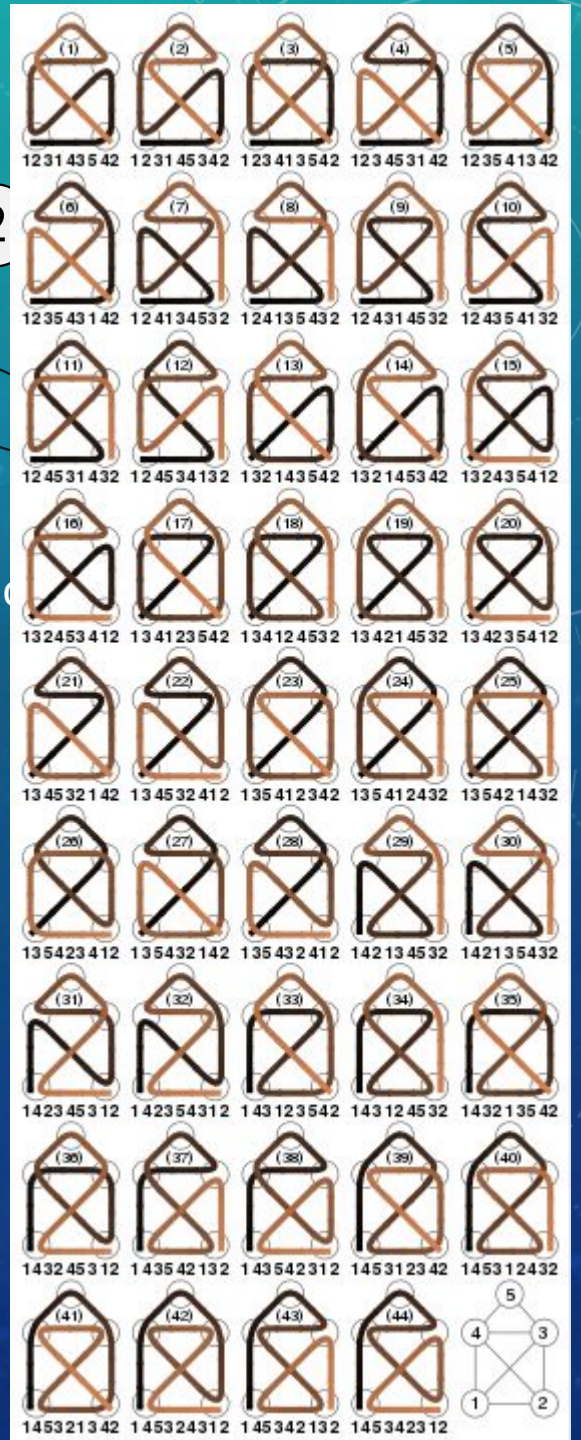
- In graph theory, the degree (or valency) of a vertex of a graph is the number of edges that are incident to the vertex
- A connected graph has an Euler cycle **<u>if and only if every vertex has even degree</u>**



Degrees are labeled for ea

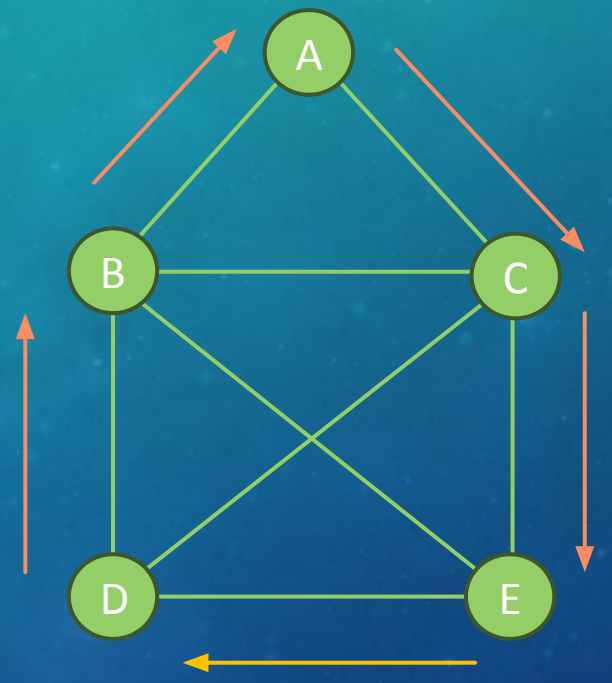


Haus vom Nikolaus



HAMILTONIAN PATH

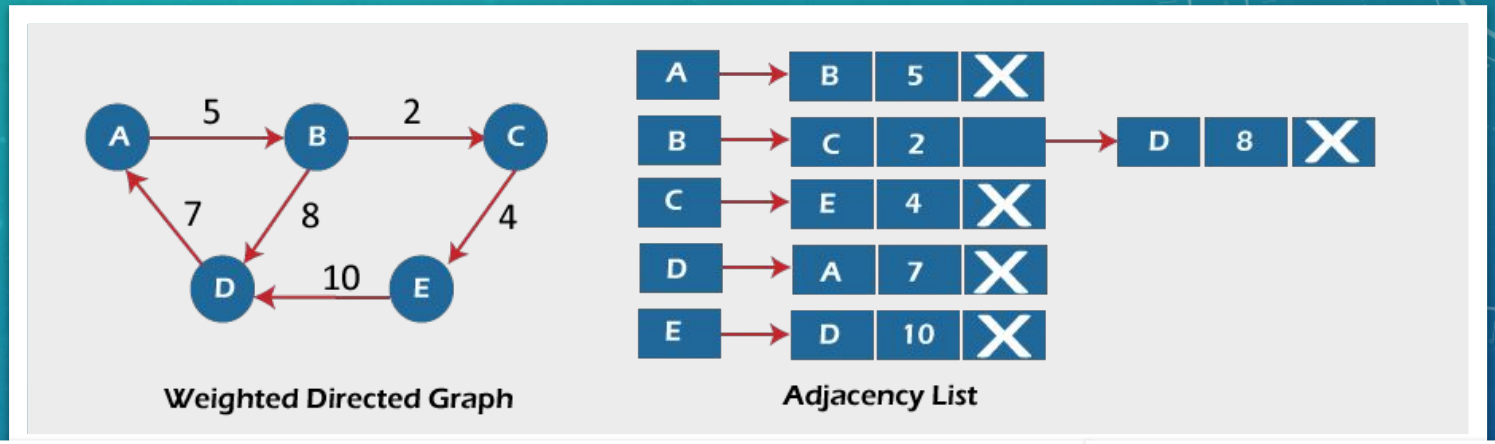
- A path in an undirected or directed graph that visits each vertex exactly once

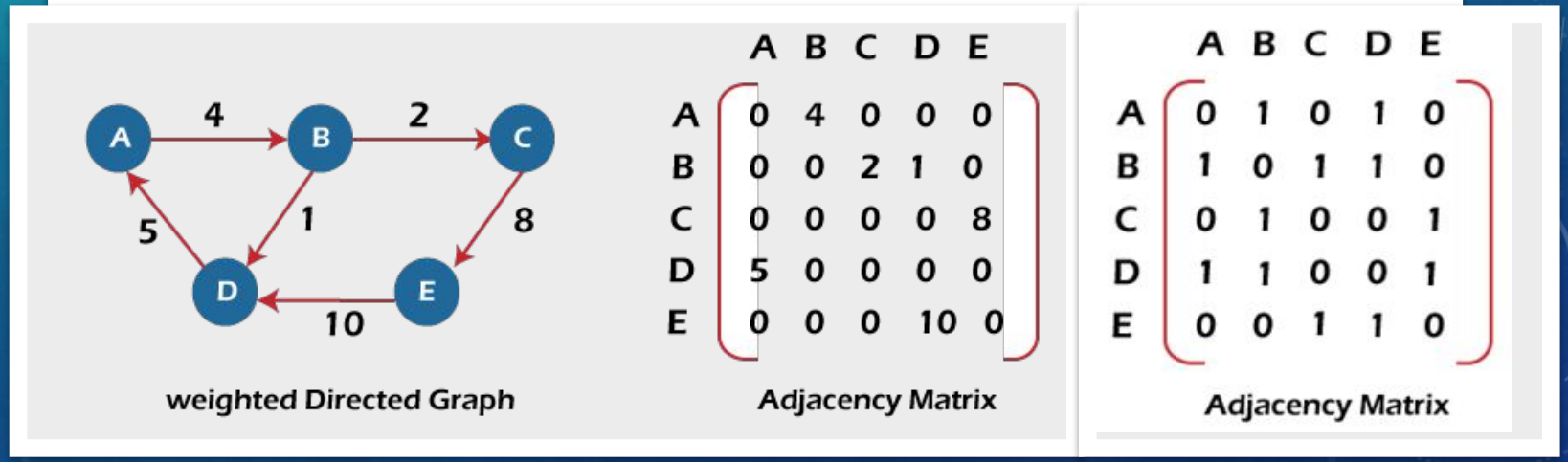


A Hamiltonian cycle: Hamiltonian path + returning to the start vertex

COMMON DATA STRUCTURES FOR GRAPH REPRESENTATION

- Adjacency list
- Adjacency matrix
- Incidence matrix (row:nodes col:edges)





https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

	Adjacency list	Adjacency matrix	Incidence matrix
Store graph	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add vertex	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Remove edge	$O(\|V\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Are vertices x and y adjacent (assuming that their storage positions are known)?	$O(\|V\|)$	$O(1)$	$O(\|E\|)$
Remarks	Slow to remove vertices and edges, because it needs to find all vertices or edges	Slow to add or remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied

STRATEGY/HIERHOLZER'S ALGORITHM (EULARIAN PATH)

Input: Undirected graph G=(V,E), no or exactly two nodes have odd degree -> Output: List of nodes in Eulerian cycle/path

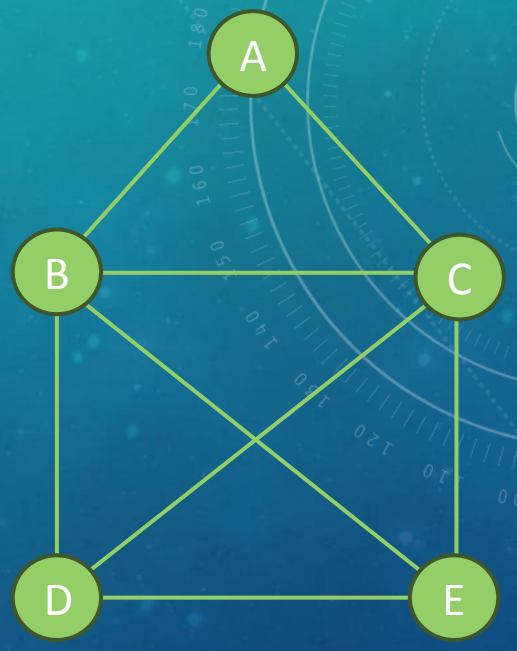
```
BEGIN
   IF graph infeasible THEN END
   IF graph semi-Eulerian THEN
      start ← node with odd degree
   ELSE
      start ← arbitrary node
   subtour ← ∅
   tour ← {start}
   REPEAT
      start ←  node in tour with unvisited edge
      subtour ← {start}
      current = start
      DO
         {current, u} ← take unvisited edge leaving current
         subtour ← subtour ∪ {u}
         current ← u
      WHILE start ≠ current
      Integrate subtour in tour
   UNTIL tour is Eulerian cycle/path
END
```

Carl Hierholzer (1840-1871)
German mathematician

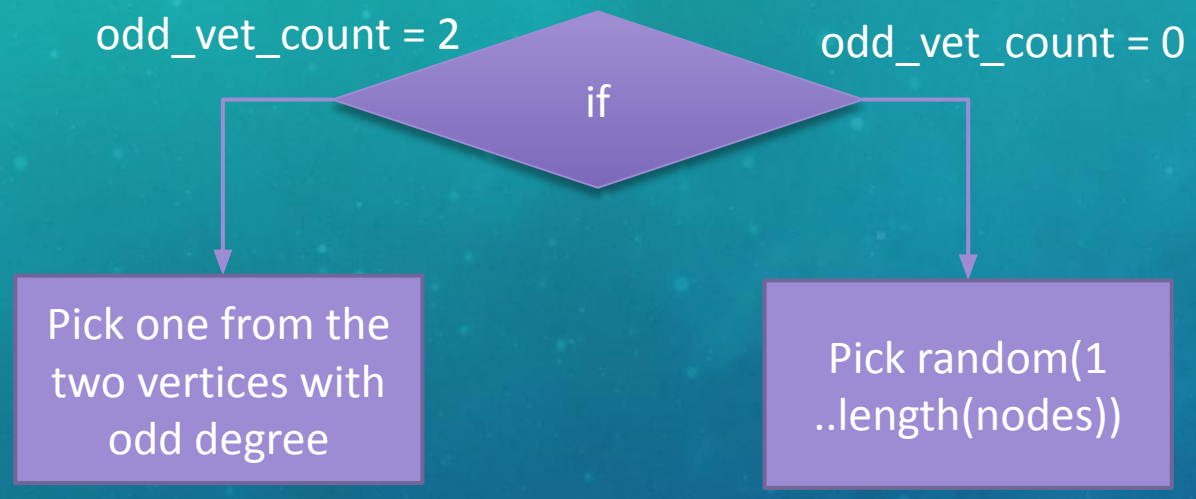
https://algorithms.discrete.ma.tum.de/graph-algorithms/hierholzer/index_en.html

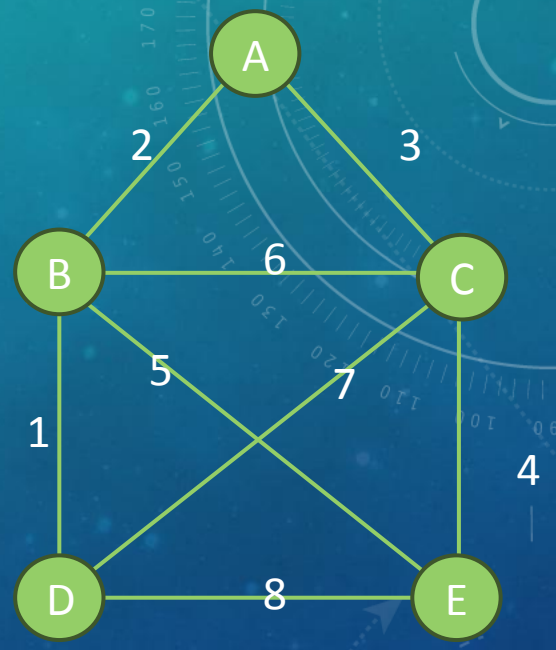
STEP 1: FEASIBILITY

```
func g_degree_count(G):
    node_list #
    degree_hash # for saving degree count for each node
    odd_vet_count = 0
    for iter in node_list:
        degree_hash{node}=degree_count(node) #degree_count
        if degree_count(node) mod 2 != 0
            odd_vet_count= odd_vet_count +1
    If odd_ver_count  > 0 && odd_ver_count  != 2 :
        return infeasible
    else:
        return feasible, odd_vet_count
    Alternative: return degree_hash, odd_vet_count
```



STEP2: IF FEASIBLE





STEP3: PATH_BUILDING

array tour # etc. data type supports pop, shift, push etc. would be the most convenient

available_edge_list = [1,2,3,4,5,6,7,8] # data type supports pop, shift, push etc. would be the most convenient

while(length(tour) <= number_of_edges)

- push picked/current vertex to tour # [D]
- List_of_con_edges_from_ava = query_edge(node, available_edge_list)
- #random pick one # e.g. we pick edge 1
- Move from D to B
- pop 1 from the available_edge_list #meaning remove the edge1 from the list of available edges
- Push edge1 to the tour #, now tour=[1]

THE SUPERSTRING PROBLEM

- Finding a <u>shortest circular</u> 'superstring' that contains **<u>all</u>** possible 'substrings' of length k (k-mers) over a given alphabet.

Nicolaas de Bruijn (1918-2012)
Dutch mathematician

- Alphabet (0,1)
- K = 3

$2^3 = 8$

000
001
010
100
101
110
011
111

The circular superstring:

$$(0001110100)_n$$

- contains all 3-mers
- as short as possible

$(0001110100)_n$

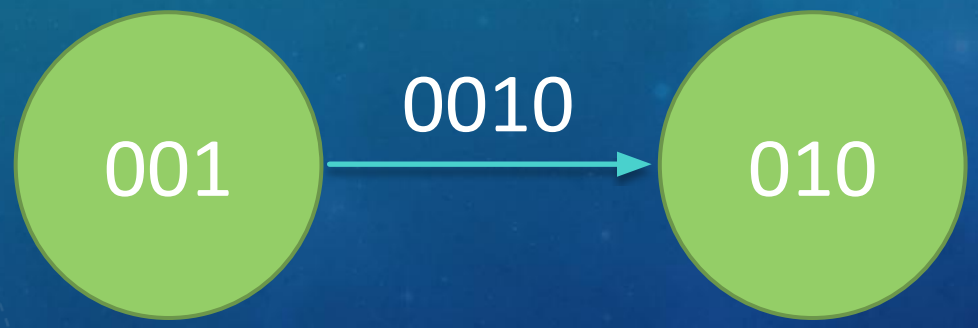
000
001
011
111
110
101
010
100

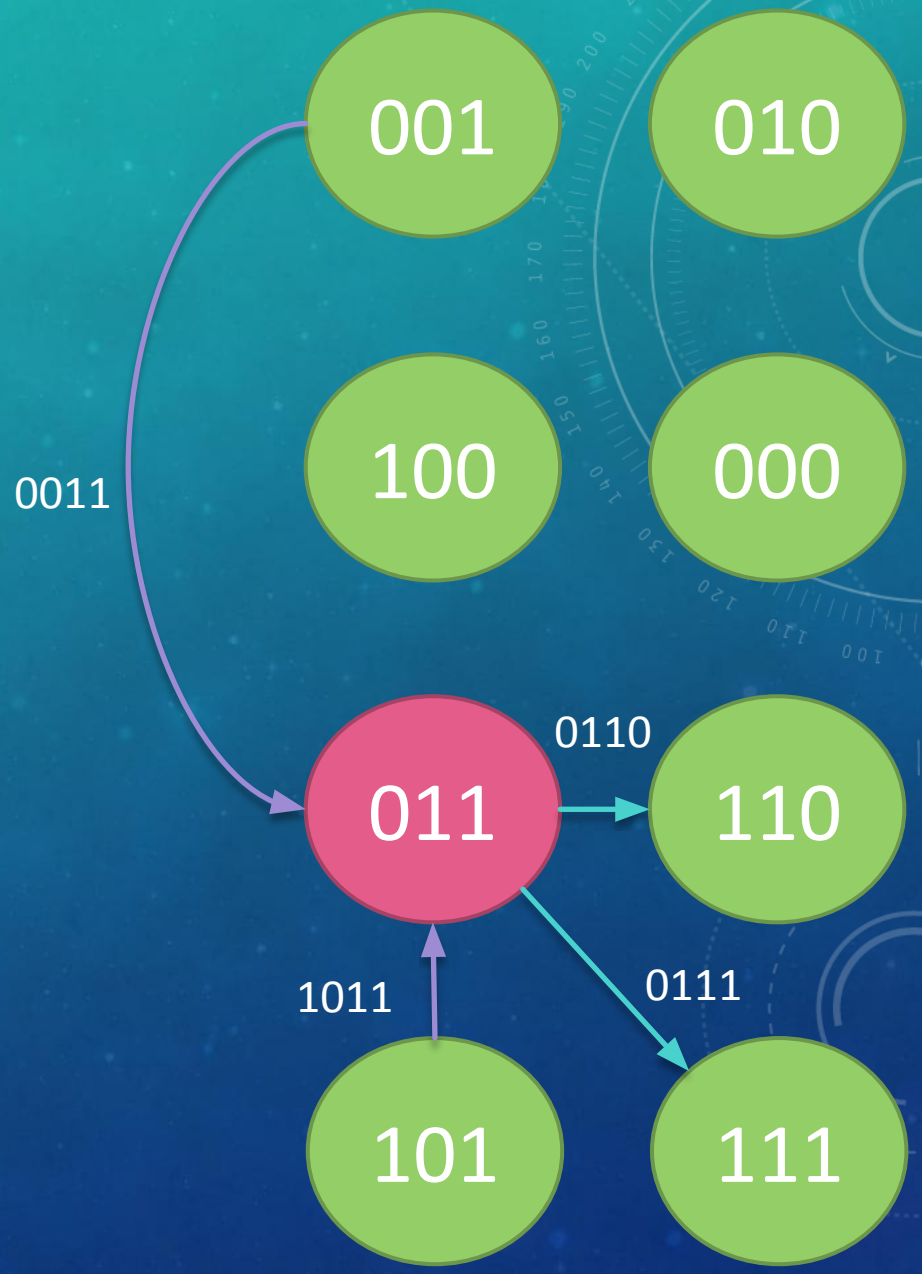
But how can one construct such a superstring for all k-mers in the case of an arbitrary value of k and an arbitrary alphabet?

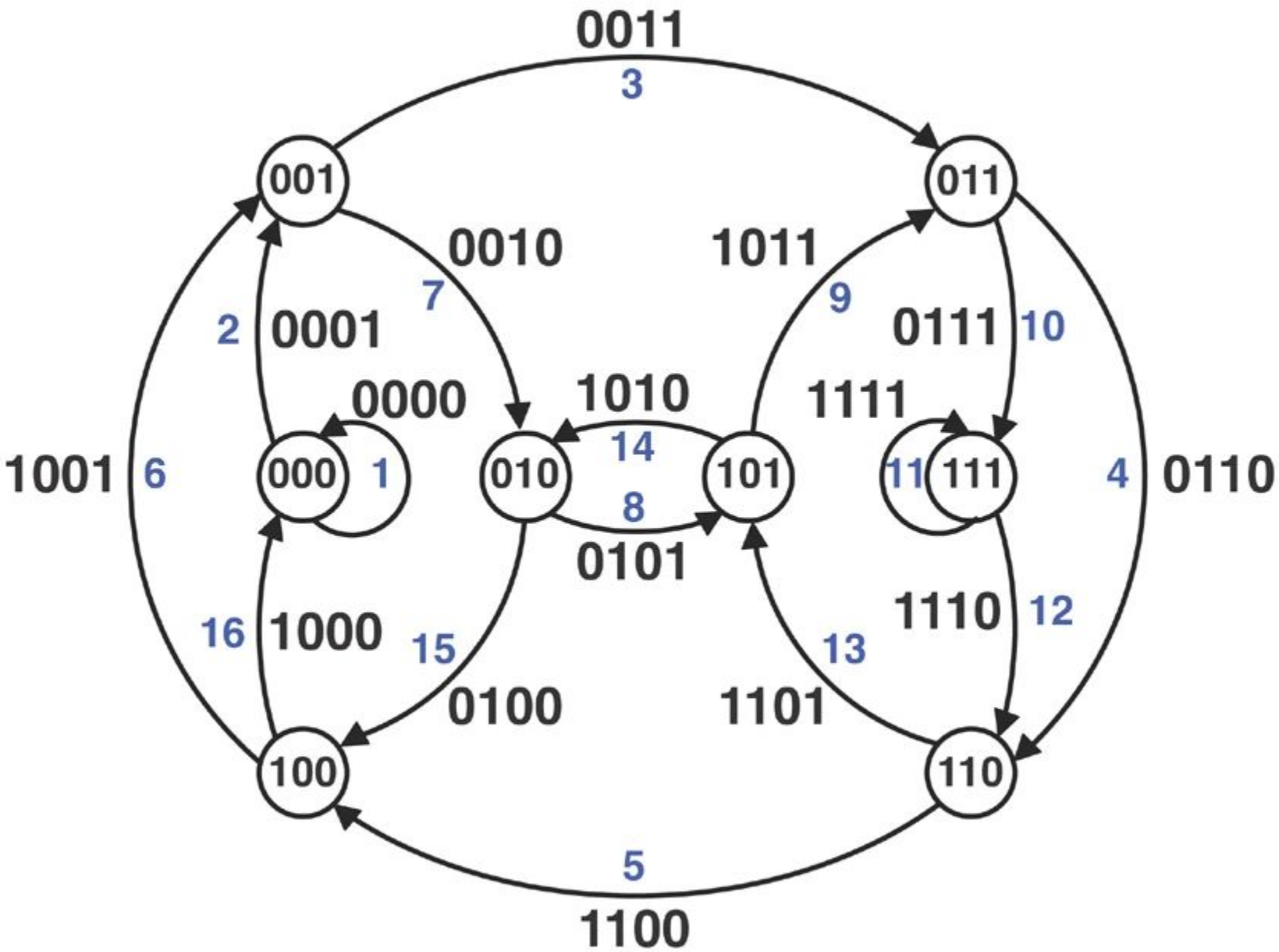
- Briefly, construct a graph *B* (the original graph called a de Bruijn graph) for which every possible ($k - 1$)-mer is assigned to a node; connect one ($k - 1$)-mer by a directed edge to a second ($k - 1$)-mer if there is some *k*-mer whose prefix is the former and whose suffix is the latter (Fig. 2). Edges of the de Bruijn graph represent all possible *k*-mers, and thus an Eulerian cycle in *B* represents a shortest (cyclic) superstring that contains each *k*-mer exactly once. By checking that the indegree and outdegree of every node in *B* equals the size of the alphabet, we can verify that *B* contains an Eulerian cycle. In turn, we can construct an Eulerian cycle using Euler's algorithm, therefore solving the superstring problem. It should now be apparent why the 'de Bruijn graph' construction described in the main text, which does not use all possible *k*-mers as edges but rather only those generated from our reads, is also named in honor of de Bruijn.

K=4, (0,1)

1. every possible (k − 1)-mer is assigned to a node
2. connect one (k − 1)-mer by a directed edge to a second (k − 1)-mer if there is some k-mer whose prefix is the former and whose suffix is the latter



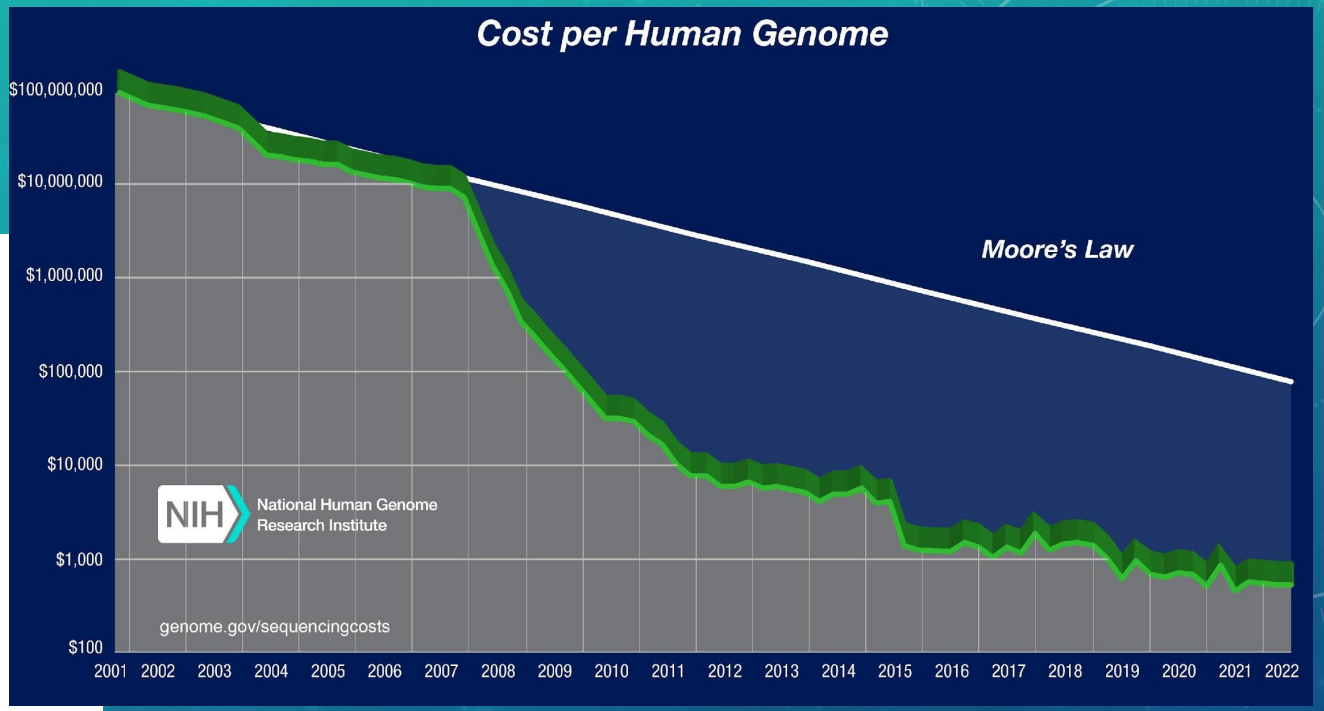


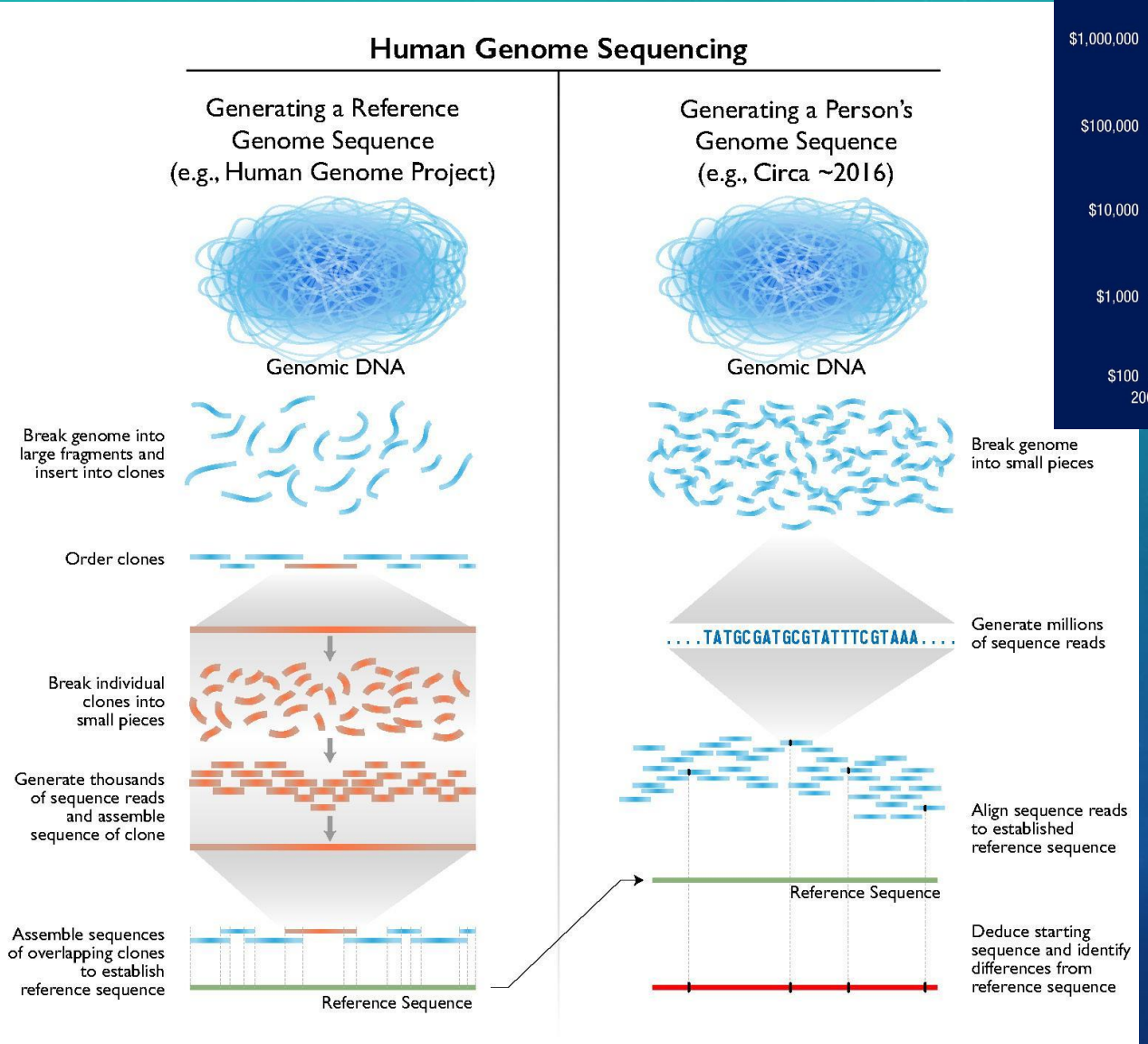


Compeau et al. 2011 *Nature Biotechnology*

- Edges of the de Bruijn graph represent **all possible *k*-mers**, and thus an Eulerian cycle in *B* represents a shortest (cyclic) superstring that contains **each *k*-mer exactly once**.
- By checking that the indegree and outdegree of every node in *B* equals the size of the alphabet, we can verify that *B* contains an Eulerian cycle. (2in2out in the example)
- In turn, we can construct an Eulerian cycle using Euler's algorithm, therefore solving the superstring problem.

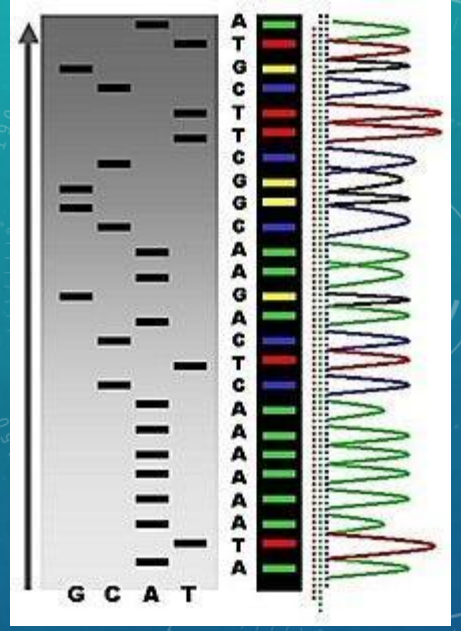
GENOME SEQUENCING



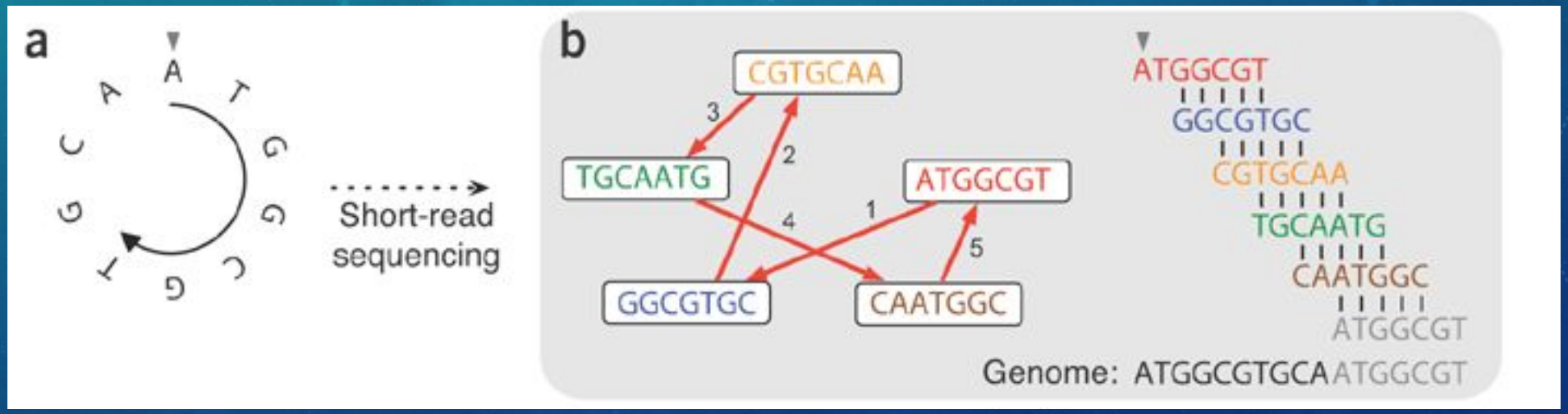


Pictures from https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost

GENOME ASSEMBLY



- Alignment based : Nodes represent the reads; Alignments between the reads are represented the edges



Compeau et al. 2011 *Nature Biotechnology*

GENOME ASSEMBLY STRATEGIES (NEXT-GENERATION SEQUENCING DATA)

- Overlap
- Layout
- Consensus

- **de Bruijn graph (DBG)**

FURTHER READING

- Miller et al. https://www.sciencedirect.com/science/article/pii/S0888754310000492

Review

Assembly algorithms for next-generation sequencing data

Jason R. Miller, Sergey Koren, Granger Sutton

J. Craig Venter Institute, 9704 Medical Center Drive, Rockville MD 20850-3343, USA

Received 4 August 2009, Accepted 2 March 2010, Available online 6 March 2010.

What do these dates mean?

Show less

Add to Mendeley Share Cite

https://doi.org/10.1016/j.ygeno.2010.03.001

Abstract

The emergence of next-generation sequencing platforms led to resurgence of research in whole genome shotgun assembly algorithms and software. DNA sequencing data from the Roche 454, Illumina/Solexa, and ABI SOLiD platforms typically present shorter read lengths, higher coverage, and different error profiles compared with Sanger sequencing data. Since 2005, several assembly software packages have been created or revised specifically for de novo assembly of next-generation sequencing data. This review summarizes and compares the published descriptions of packages named SSAKE, SHARCGS, VCAKE, Newbler, Celera Assembler, Euler, Velvet, ABySS, AllPaths, and SOAPdenovo. More generally, it compares the two standard methods known as the de Bruijn graph approach and the overlap/layout/consensus approach to assembly.

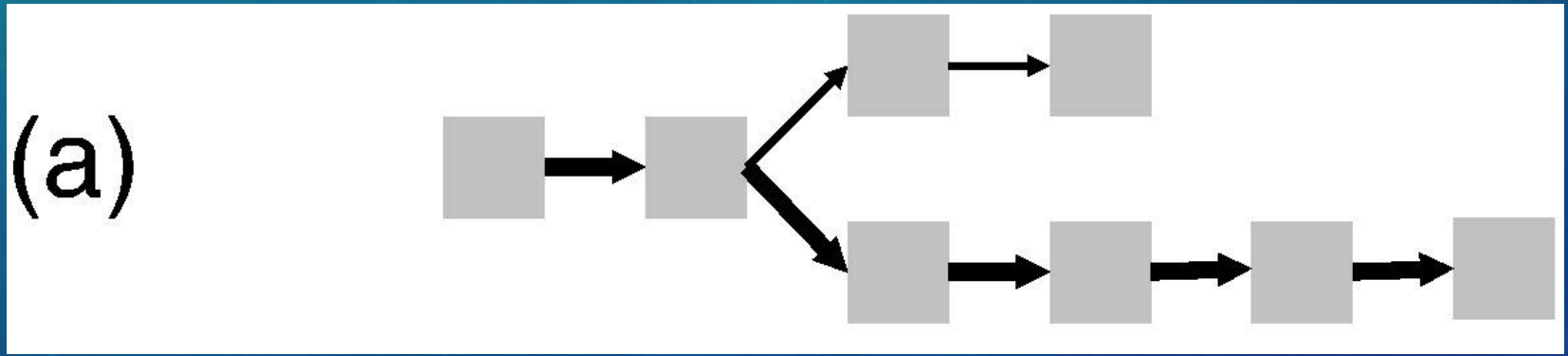
WHY K-MER?

- Practical problems
  - ❑ Reads are not error free.
  - ❑ The coverage is not always uniform across the genome.
  - ❑ Genome could be repetitive
  - ❑ The genome may not be a single circular chromosome (multiple & linear)

REAL-WORLD WGS

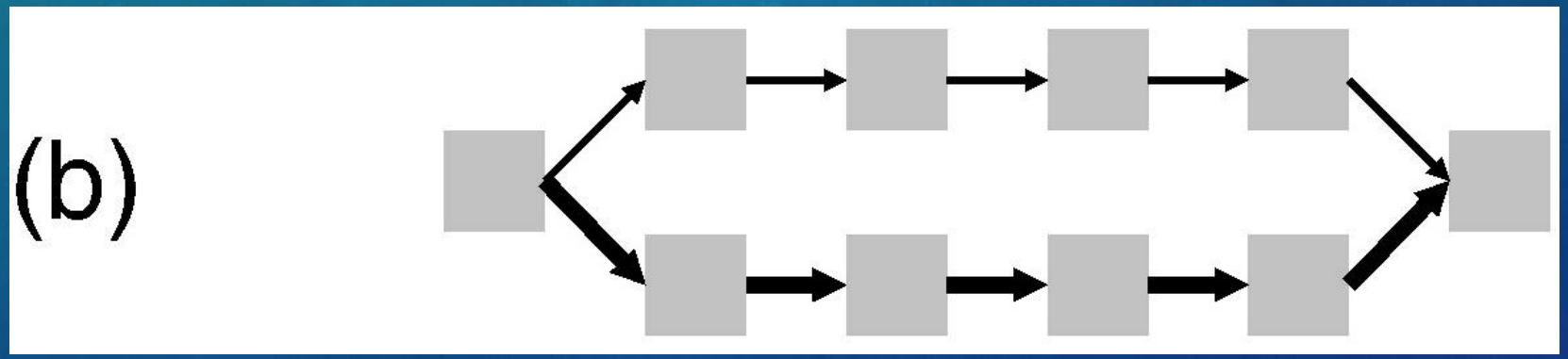
Miller et al. 2010 *Genomics*

- Spurs are short, dead-end divergences from the main path (Fig. 3a). They are induced by sequencing error toward one end of a read. They can be induced by coverage dropping to zero.



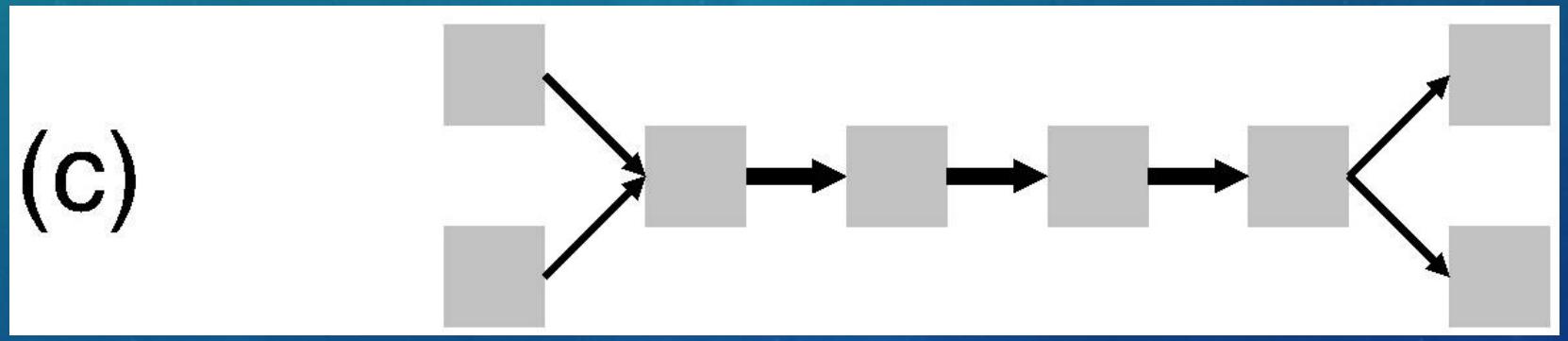
REAL-WORLD WGS

- Bubbles are paths that diverge then converge (Fig. 3b). They are induced by sequencing error toward the middle of a read, and by polymorphism in the target. Efficient bubble detection is non-trivial



REAL-WORLD WGS

- Paths that converge then diverge form the frayed rope pattern (Fig. 3c). They are induced by repeats in the target genome.

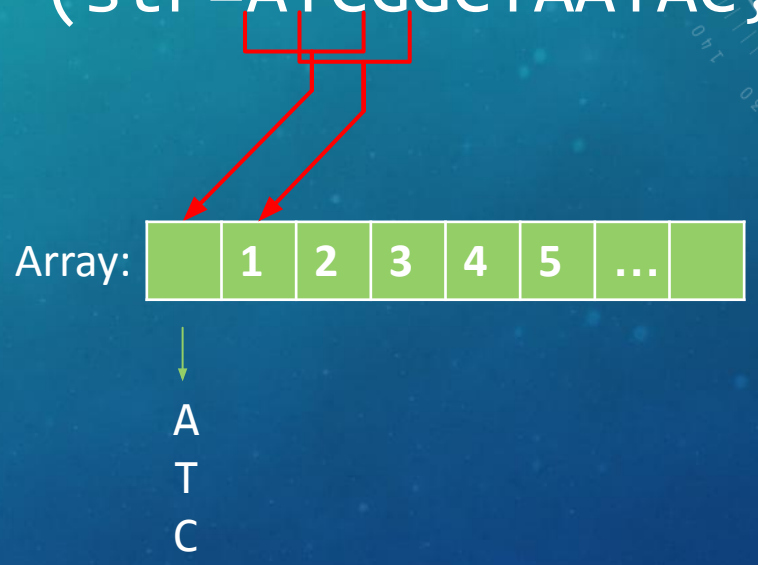


Cycles are paths that converge on themselves. They are induced by repeats in the target. For instance, short tandem repeats induce small cycles.

IMPLEMENTATION OF KMER (STEP 1 EXTRACTION)

```
Kmer_extract(str Input_String, int k):
    L = length(Input_String)
    arr #New array or hash for empty
    for(n=0;n+k<L;n++){
        arr[n] = substr(Input_String,k)
    }
    return arr
```

(str=ATCGGCTAATAC, k=3)

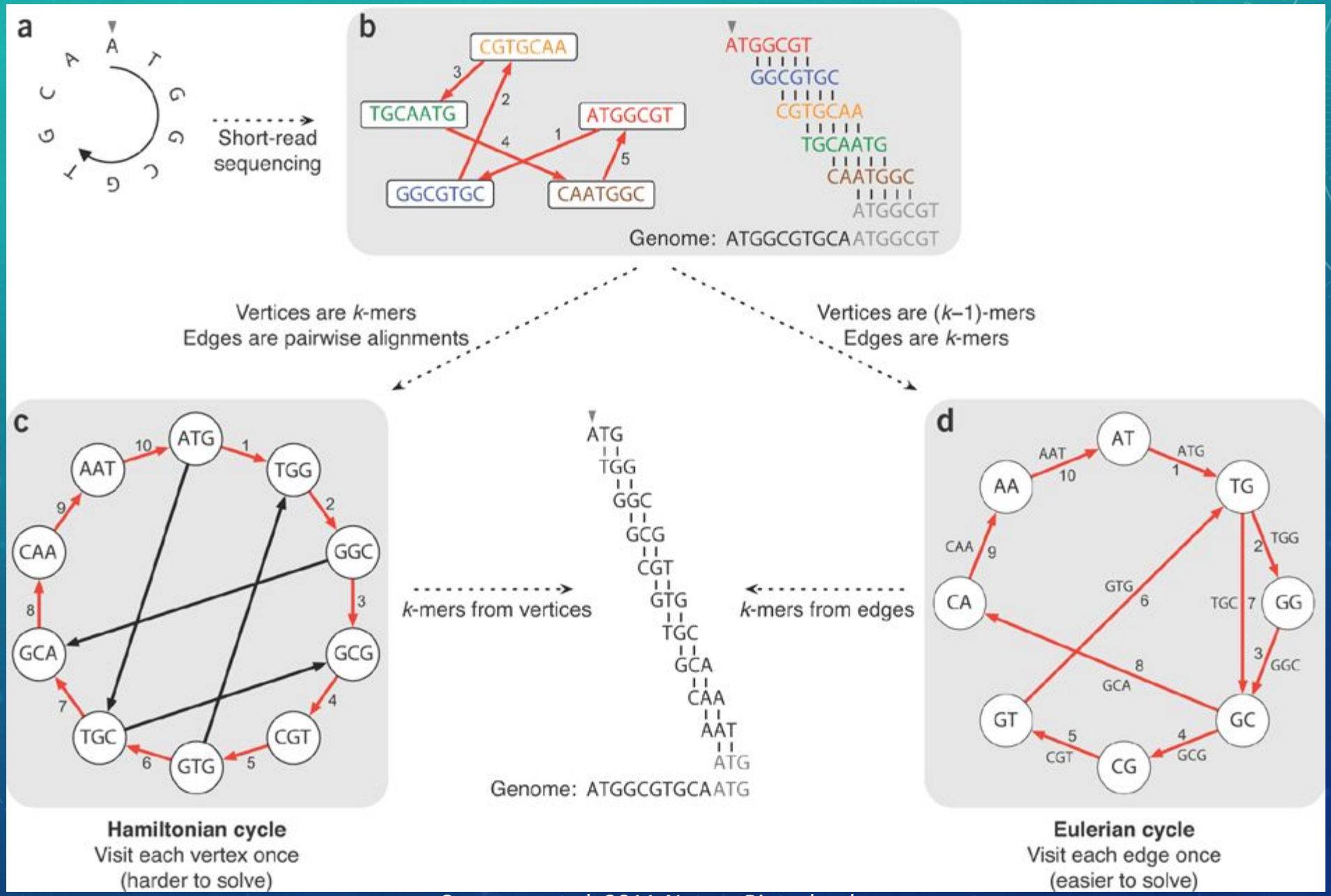


IMPLEMENTATION OF KMER (STEP 2 SIMPLET COUNTING)

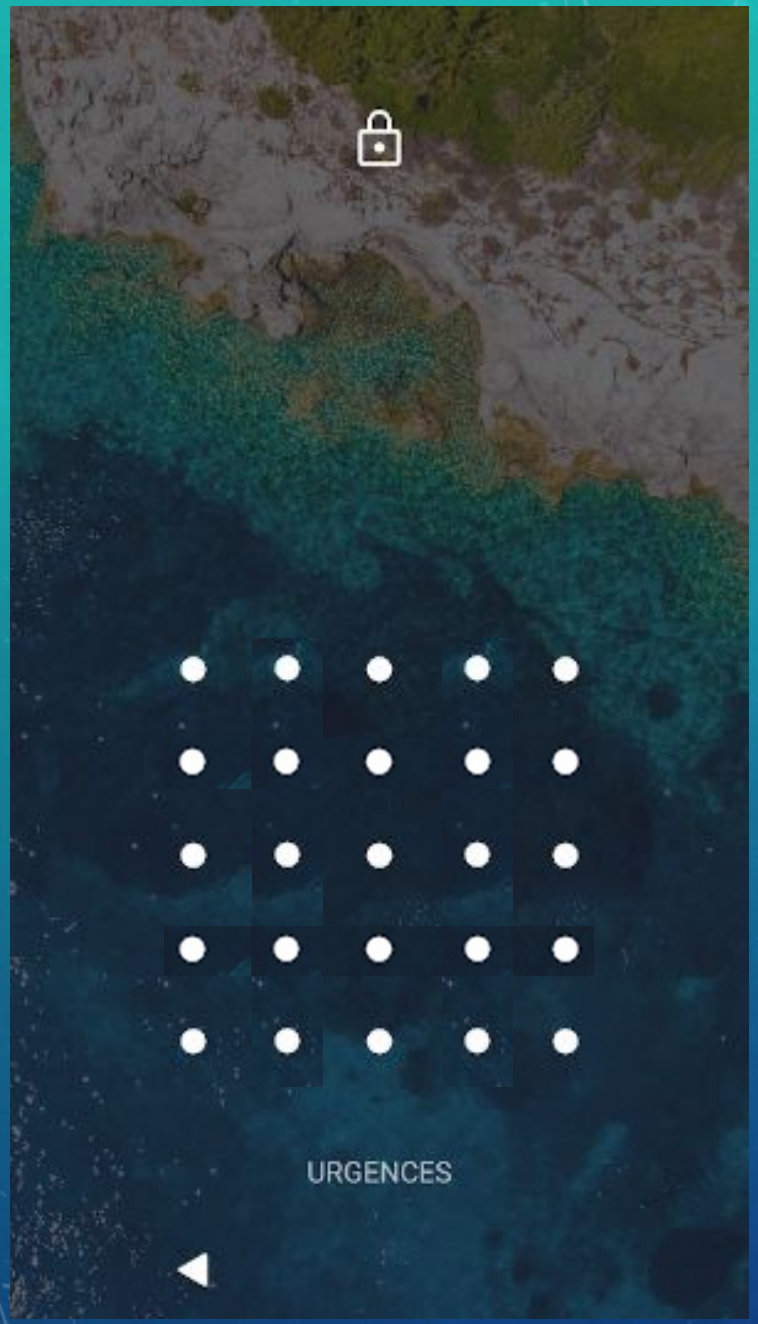
```
Kmer_count(array):
    L = length(array)
    kmer_count #New hash for counting
    for(n=0;n<L;n++){
        if array[n] in kmer_count{array[n]}:
            kmer_count{array[n]}=kmer_count{array[n]}+1
        else:
            kmer_count{array[n]}=1
    }
    return kmer_count
```

KMER SOFTWARE

- JellyFish for *k*-mer counting



Compeau et al. 2011 *Nature Biotechnology*



- ChatGPT: The NP-completeness of the Hamiltonian path problem suggests that as the size of the graph increases, the time required to find a Hamiltonian path grows exponentially, making it computationally more challenging compared to the polynomial-time solution available for the Eulerian path problem.

REMARKS

FUTURE

- The Breadth-First search algorithm (BFS)
- Depth-first search (DFS)
- Shortest path search (Directions in a map)
- Many more …