ALGORITHMS OF BIOINFORMATICS

4 Assembling Genomes

20 November 2025

Prof. Dr. Sebastian Wild

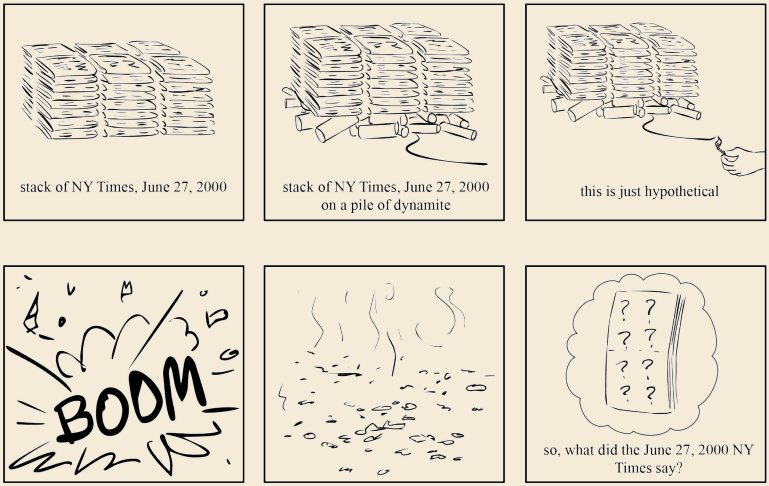
CS594 (Winter 2025-26)
Philipps-Universität Marburg
version 2025-11-27 21:27 H

Outline

4 Assembling Genomes

4.1 Exploding Newspapers

Exploding Newspapers



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.1

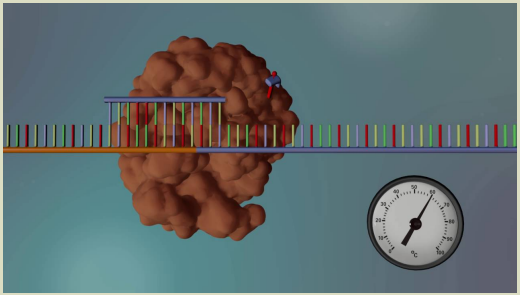
https://cogniterra.org/lesson/29884/step/2?unit=21982

DNA Sequencing

How can we possibly "read out" DNA sequences?

Through clever combinations of chemistry and engineering . . .

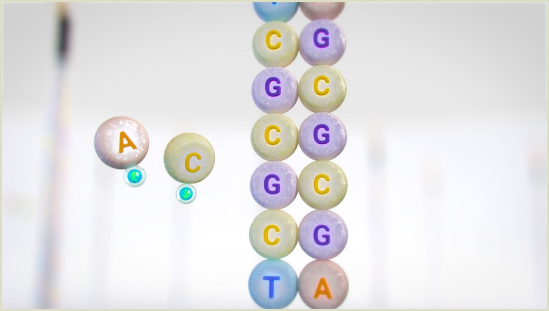
- Historically first method: *Sanger sequencing*
  - developed 1977 by Frederick Sanger
  - earning him his *second* Nobel prize in chemistry
    (the first one was for determining the amino acid sequence of insulin)



DNA Sequencing - 3D
https://youtu.be/ONGdehkB8jU

“Next Generation Sequencing”

- group of methods that are massively parallel
- ⇝ easier to automate, much cheaper per read
- commercially available since 2005



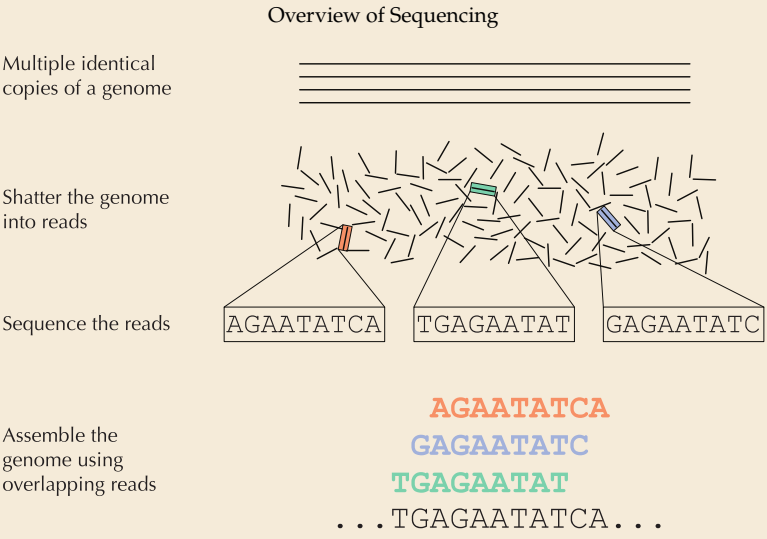
Overview of Illumina Sequencing by Synthesis Workflow
https://youtu.be/EDVKxSNdSic

Limitations fo Sequencing Techniques

- Sequencing machines can produce somewhat reliable *reads* of a few hundred to a few thousand bases.

$\rightsquigarrow$ NGS will produce many reads, covering much of sequence but we need to *assemble* entire DNA sequence

Overview



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig. 4.13
https://cogniterra.org/lesson/29918/step/2?unit=22015

In the following: Consider an idealized situation.

4.2 The De-Novo Sequencing Problem

Formalization of Sequence Assembly

- k-mer = length-k-(sub)string
- Given text (genome) $T[0..N)$ and k,
  the *k-mer composition* of T is the multiset of all k-mers in T:
  $\text{Composition}_k(T) \;=\; \{\!\{T[0..k), T[1..1+k), T[2..2+k), \ldots, T[n-k, n)\}\!\}$
- Example: $\text{Composition}_3(\texttt{TATGGGGTGC}) = \{\!\{\texttt{ATG}, \texttt{GGG}, \texttt{GGG}, \texttt{GGT}, \texttt{GTG}, \texttt{TAT}, \texttt{TGC}, \texttt{TGG}\}\!\}$
- **(De-Novo) Sequence Assembly Problem:**
  - Given multiset of k-mers $\mathcal{R}$
  - Find a string T such that $\text{Composition}_k(T) = \mathcal{R}$
- Example: $\mathcal{R} = \{\!\{\texttt{AAT}, \texttt{ATG}, \texttt{GTT}, \texttt{TAA}, \texttt{TGT}\}\!\}$
  - form overlapping pairs
  - T must start with `TTA` as no 3-mer ends with `TA` and end with `GTT`

```
TAA
 AAT
  ATG
   TGT
    GTT
TAATGTT
```

. . . not always as easy

Now consider $\mathcal{T} = \{\{\texttt{AAT}, \texttt{ATG}, \texttt{ATG}, \texttt{ATG}, \texttt{CAT}, \texttt{CCA}, \texttt{GAT}, \texttt{GCC}, \texttt{GGA}, \texttt{GGG}, \texttt{GTT}, \texttt{TAA}, \texttt{TGC}, \texttt{TGG}, \texttt{TGT}\}\}$

Let's start again with TTA.
The first few steps are forced.

```
TAA
 AAT
  ATG
TAATG
```

Then we have there are 3 options TG[CGT].
Say we pick TGT

```
TAA
 AAT
  ATG
   TGT
    GTT
TAATGTT ϟ
```

We've reached a dead end!

If we instead choose more wisely,
we can continue:

```
TAA
 AAT
  ATG
   TGC
    GCC
     CCA
      CAT
       ATG
        TGG
         GGG
          GGA
           GAT
            ATG
             TGT
              GTT
TAATGCCATGGGATGTT
```

Escaping the Dead Ends

How can we systematically avoid having to try all options?

Repeats make life challenging.

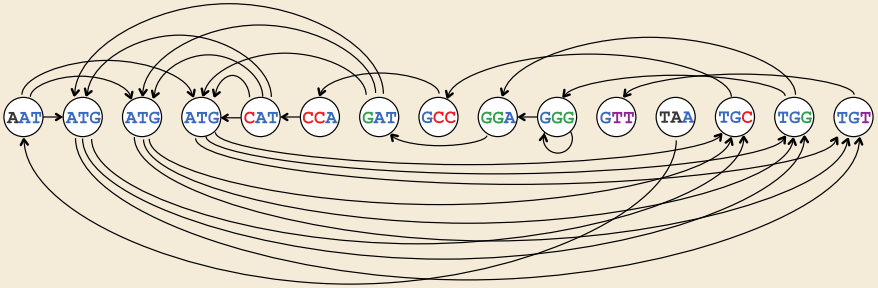
4.3 Assembly with Overlap Graphs

Read Overlap Graphs

Given a k-mer composition $\mathcal{R} = \{\{R_0, \ldots, R_{n-k}\}\}$,
construct the *overlap graph* as directed graph $G_O(\mathcal{R}) = (V, E)$

- $V = \{r_0, \ldots, r_{n-k}\}$ with r_i labeled with R_i
  can have duplicate k-mers
- $r_i r_j \in E$ if $R_i[1..k) = R_j[0..k-1)$

Example Overlap Graph (Alphabetical)

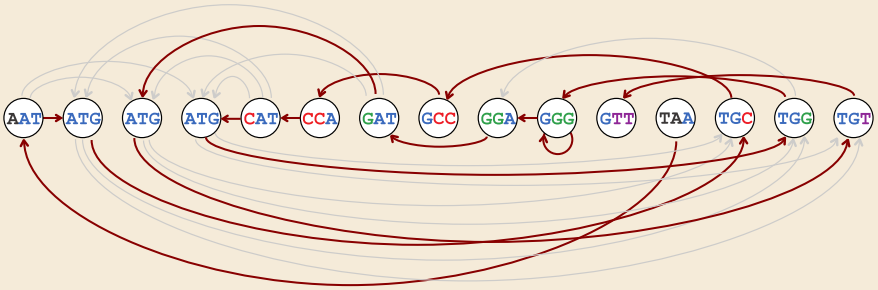


Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.8
https://cogniterra.org/lesson/29886/step/7?unit=21984

Hamiltonian Paths

Given the overlap graph $G(\mathcal{R})$ of a k-mer composition $\mathcal{R}$
a sequence assembly solution is a *Hamiltonian path* in $G(\mathcal{R})$

Hamiltonian Path in Overlap Graph



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.9
https://cogniterra.org/lesson/29886/step/8?unit=21984

- **Problem 1:** Result may not unique (as in this example)
- **Problem 2:** Directed Hamiltonian Path is NP-complete.

Bad News Again

It seems we once again ran into a hard problem.

(And we haven't even accounted for computing the overlap graph . . .)

. . . but sometimes, a different way to look at the problem helps.

4.4 De Brujin Graphs

Origins

Nicolaas Govert de Bruijn's *k-universal string puzzle*:

How long does a binary string need to be to contain all 2^k binary k-mers as substrings?
How to construct a shortest such string?

Examples:

- $k = 2$
  - must contain `00`, `01`, `10`, `11`
  - $\rightsquigarrow$ a 2-universal string needs at least 5 letters
    4 starting positions, each for a 2-letter substring
  - Also achievable: `00110`
  - $\rightsquigarrow$ Answer: 2-universal requires 5 characters.
- $k = 3$ $\rightsquigarrow$ a shortest 3-universal string is `0001110100`

Same question makes sense for strings over alphabet $\Sigma = [0..\sigma)$.
$\rightsquigarrow$ T is k-universal $\iff$ contains all σ^k k-mers as substring.

de Bruijn Graphs

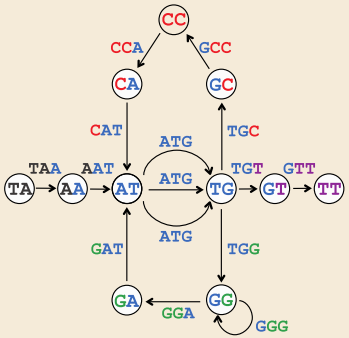
Given a k-mer composition $\mathcal{R} = \{\!\{R_0, \ldots, R_{n-k}\}\!\}$, construct the *de Bruijn graph* as directed **multi**graph $G_B(\mathcal{R}) = (V, E)$

- $V = \{R[0..k-1) : R \in \mathcal{R}\}$
- For each read $R \in \mathcal{R}$, add the edge vw (with label $R[0..k)$) to E
  for $v = R[0..k-1)$ and $w = R[1..k)$

$\rightsquigarrow$ Sequence using each k-mer once = Euler path in G_B

- Euler path efficiently computable!

DeBruijn$_3$(**TAATGCCATGGGATGTT**)



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig. 4.1
https://cogniterra.org/lesson/29910/step/2?unit=22007

k-universal strings

The de Bruijn graph "solves" the universal string problem for $\mathcal{R}$ all σ^k k-mers

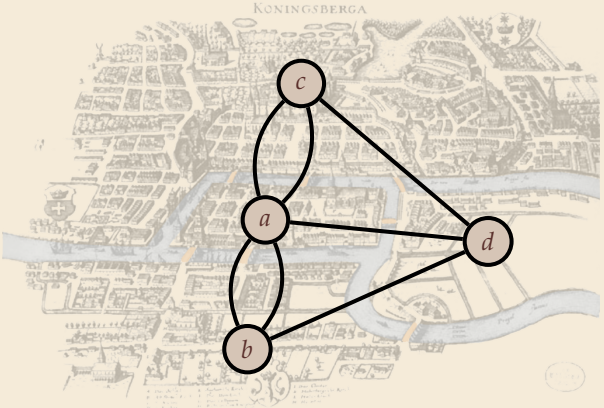
- σ^{k-1} vertices, σ^k edges
- each vertex has out- and in-degree $= \sigma$

$\rightsquigarrow$ G_B is Eulerian

$\rightsquigarrow$ Compute Euler path (see below)

Euler Cycles

Euler Walk: Walk using every edge in $G = (V, E)$ exactly once.



Euler's Theorem:
Euler walk exists iff G connected and 0 or 2 vertices have odd degree.

'$\Rightarrow$' trivial (need to enter and exit intermediate vertices equally often)

'$\Leftarrow$' Following algorithm *constructs* Euler walk under this assumption

Euler Cycles – Hierholzer's Algorithm

- use an *edge-centric DFS*
  - We mark *edges* (not vertices)
  - ⇝ stack = **edge-simple walk**
  - We remember iterator i globally per v to resume traversal

```
1  procedure eulerWalk(G):
2      // Assume G = (V, E) is connected (multi)graph
3      V_odd := {v ∈ V : d(v) odd}
4      if |V_odd| ∉ {0, 2} return NOT_EULERIAN
5      if V_odd = {x, y} then s := x else s := 0
6      euler[0..m) := NONE;  j := m − 1
7      visited[0..n, 0..n) := false // mark edges as visited
8      for v := 0, ..., n − 1
9          // globally remember next unexplored edge
10         nextEdge[v] := G.adj[w].iterator())
11     edgeDFS(s)
12     return euler
```

```
1  procedure edgeDFS(s):
2      frontier := new Stack;
3      frontier.push(s)
4      while ¬frontier.isEmpty()
5          v := frontier.top();  i := nextEdge[v]
6          if ¬i.hasNext() // v has no unused edge
7              frontier.pop()
8              if ¬frontier.isEmpty()
9                  // assign edge leading here largest free index
10                 euler[j] := (frontier.top(), v);  j := j − 1
11             end if
12         else
13             w := i.next()
14             if ¬visited[v, w]
15                 visited[v, w] := true
16                 visited[w, v] := true
17                 frontier.push(w)
18             end if
19         end if
20     end while
```

Directed Euler Cycles

- de Bruijn graphs are *directed* . . . what changes?
- **Euler's Theorem, directed version:**
  Euler walk exists iff G is (a) strongly connected and
  (b) $d_{\text{out}}(v) - d_{\text{in}}(v) = 0$ for all vertices v or
  0 for all but 2 vertices where it is $+1$ and -1, respectively.
  - '$\Rightarrow$' need to enter and exit intermediate vertices equally often
  - '$\Leftarrow$' Hierholzer's algorithm also works on directed graphs, with changes below.
- Changes to Hierholzer's algorithm:
  - Check balance ($d_{\text{out}}(v) - d_{\text{in}}(v)$) of vertices
  - start at the unique vertex with $d_{\text{out}}(s) - d_{\text{in}}(s) = +1$ if s exists; otherwise arbitrary
  - in edgeDFS, do not mark the reverse edge as also visited

Space-Efficient Hierholzer

frontier can use up to $O(m)$ extra space (v can appear $d(v)$ times); can we avoid that?

Space-Efficient Hierholzer:
Eulerian Cycles in $O(m)$ Time and $O(n)$ Space*

Ziad Ismaili Alaoui[1] Detlef Plump[2] Sebastian Wild[3]

Abstract. We describe a simple variant of Hierholzer's algorithm that finds an Eulerian cycle in a (multi-)graph with n vertices and m edges using $O(n \lg m)$ bits of working memory. This substantially improves the working space compared to standard implementations of Hierholzer's algorithm, which use $O(m \lg n)$ bits of space. Our algorithm runs in linear time, like the classical versions, but avoids an $O(m)$-size stack of vertices or storing information for each edge. To our knowledge, this is the first linear-time algorithm to achieve this space bound, and the method is very easy to implement. The correctness argument, by contrast, is surprisingly subtle; we give a detailed formal proof. The space savings are particularly relevant for dense graphs or multigraphs with large edge multiplicities.

1 Introduction. An *Eulerian cycle* in a graph is a closed walk that traverses every edge exactly once. A graph is called *Eulerian* if it contains an Eulerian cycle. Euler's study of the existence of Eulerian cycles and their computation is often cited as the birth of modern graph theory. Apart from historical significance, efficiently computing them has applications in, e.g., genome assembly [7], routing problems [6], and 3D printing [9].

Classic algorithms such as Fleury's algorithm or Hierholzer's algorithm [5], both dating from the late 19th century, are simple to describe and find an Eulerian cycle in any Eulerian graph in polynomial time. Indeed, both consider each edge only once, and Hierholzer's algorithm can be implemented to run in optimal linear time. Being both simple and efficient, the latter is the method of choice in practice. Typical implementations of Hierholzer's algorithm (see below) use $O(m \lg n)$ bits of working memory. (Here and throughout, $\lg = \log_2$.)

In this paper, we present a simple variant of Hierholzer's algorithm that uses only $O(n \lg m)$ bits of working memory and runs in optimal $O(n + m)$ time. For dense graphs or multigraphs with many parallel edges, this is a substantial saving. Note that this space can neither hold the entire input nor the entire output of the problem; we treat the input graph as given in read-only memory and produce the Eulerian cycle *in order* to a write-only stream. In particular, our algorithm would be suitable for an application that directly consumes and uses edges of the Eulerian cycle as they are computed, potentially without ever storing the entire cycle. By working memory, we mean the extra space occupied by auxiliary data structures during the computation. Our algorithm relies on an assumption about the graph representation, which is satisfied in typical adjacency-list implementations: for directed graphs, we require that we can iterate over incoming and outgoing edges; for undirected graphs, we (instead) require a consistent ordering of vertices across all adjacency lists.

To our knowledge, we present the first algorithm with working space $O(n \lg m)$ bits that runs in linear time and explicitly produces an Eulerian cycle as its output. Our algorithm is also usable in rule-based models of computation, such as graph-transformation languages like GP 2 [8], which do not natively support stacks or recursion. This is not known to be true for other standard efficient implementations of Eulerian cycle algorithms.

1.1 Hierholzer's Algorithm. Hierholzer's algorithm was originally described in 1873 [5], unsurprisingly without detailed information about data structures for efficient execution on a computer. The original algorithm consists of following an arbitrary walk in an Eulerian graph until we get stuck, which can only happen when we closed a cycle. If this cycle has not used all edges yet, starting at such an unused edge, again following an arbitrary walk of unused edges can again only get stuck when closing a cycle; we can then fuse this new cycle into

*Ziad Ismaili Alaoui and Sebastian Wild are supported by EPSRC grant EP/X030407/1.
[1]University of Liverpool, United Kingdom (ziad.ismaili-alaoui@liverpool.ac.uk).
[2]University of York, United Kingdom (detlef.plump@york.ac.uk).
[3]Philipps-Universität Marburg, Germany, and University of Liverpool, United Kingdom (wild@informatik.uni-marburg.de).

SOSA 2026

Insight: Euler walk doesn't require full edge DFS
(backtracking edges in reverse order of forward traversal)
Suffices to not get stuck!

edges initially **Black**
double line = current vertex

We do a *reverse* edge DFS ⇝ Euler walk in correct order
Red = DFS tree edges
vertex already seen ⇝ mark **Green**
need to backtrack? ⇝ prefer **Green**

whenever we backtrack, mark Dashed and output edge

Space-Efficient Hierholzer – Implementing colors

Assumption: Can iterate over outgoing **and** incoming edges

- Every vertex has at most one outgoing **RED** edge

$\rightsquigarrow$ store explicitly in $O(n)$ space

- edges are visited (**RED** or **GREEN**) iff before current iterator in adjacency list

$\rightsquigarrow$ stored implicitly!

Back to Genomes

- Eulerian walk usually not unique or doesn't exist!

⇝ often stop with assembling *contigs*, long contiguous substrings of the genome to be further processed

- contigs can be found from de Bruijn graph using vertex balances

4.5 Practical Assemblers

Practical Complications

Challenges for DNA sequencing in practice

- DNA is *not* a single string; one per chromosome!
- we have (most) chromosomes twice (mom's and dad's), probably similar in large parts!
- imperfect coverage: some regions might not be covered by any/enough reads
- sequencing isn't exact; small reading errors need to be tolerated
- DNA is double stranded $\rightsquigarrow$ get reads from both strands mixed up

Some solutions

Imperfect coverage

- Rather unlikely to see a read for every possible starting position.
- But this is harmless; we can break reads of length ℓ into artificial k-mers for some $k < \ell$ and obtain perfect coverage!

Chromosome pairs

- problematic because of high similarity
- possible in lab to avoid sequencing both

Inexact Overlaps seem to be the most problematic.

There's life in the old dog yet!

- Inexact reads actually much easier to handle in **overlap graph**
  weighted edges for length / similarity of overlap
- highly effective heuristic: preprocessing overlap graph to transitive reduction
- Hamiltonian path / TSP tour not the bottleneck "hard in theory, easy in practice"
- challenge there: how to compute overlap graph (edge weights) efficiently
  - ⇝ we will come back to this