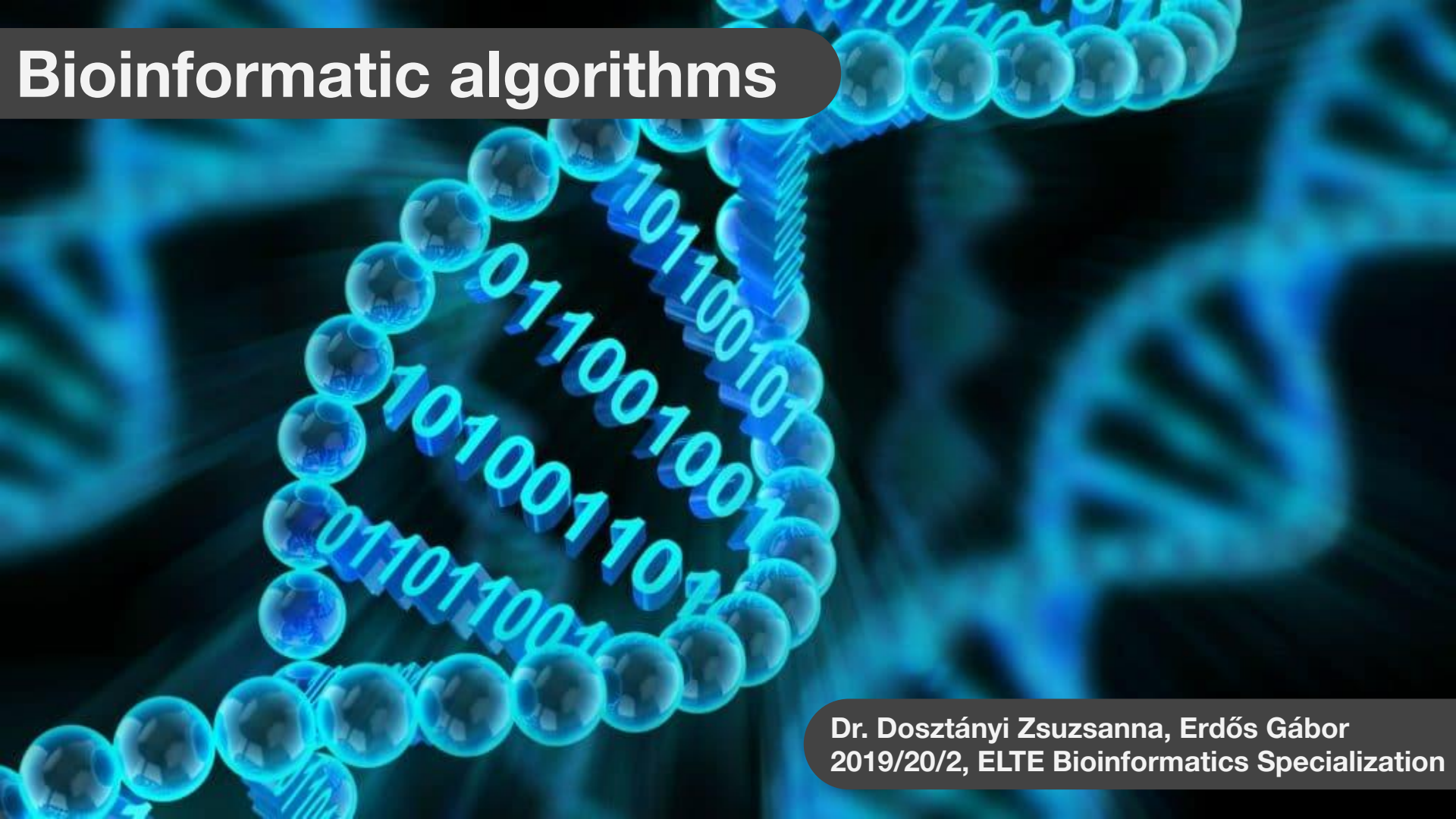
Bioinformatic algorithms



Dr. Dosztányi Zsuzsanna, Erdős Gábor
2019/20/2, ELTE Bioinformatics Specialization

The Overlap Graph Problem

Solve The Overlap Graph Problem

Input: A collection *Patterns* of *k*-mers.

Output: The overlap graph *Overlap*(*Patterns*), in the form of an adjacency list.

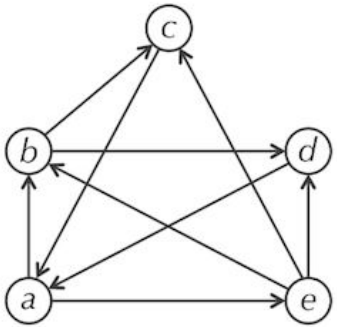
Sample Input:

```
ATGCG
GCATG
CATGC
AGGCA
GGCAT
GGCAC
```

Sample Output:

```
CATGC -> ATGCG
GCATG -> CATGC
GGCAT -> GCATG
AGGCA -> GGCAC,GGCAT
```

Graph



Adjacency Matrix

	a	*b*	*c*	*d*	*e*
a	0	1	0	0	1
b	0	0	1	1	0
c	1	0	0	0	0
d	1	0	0	0	0
e	0	1	1	1	0

Adjacency List

a is adjacent to *b* and *e*

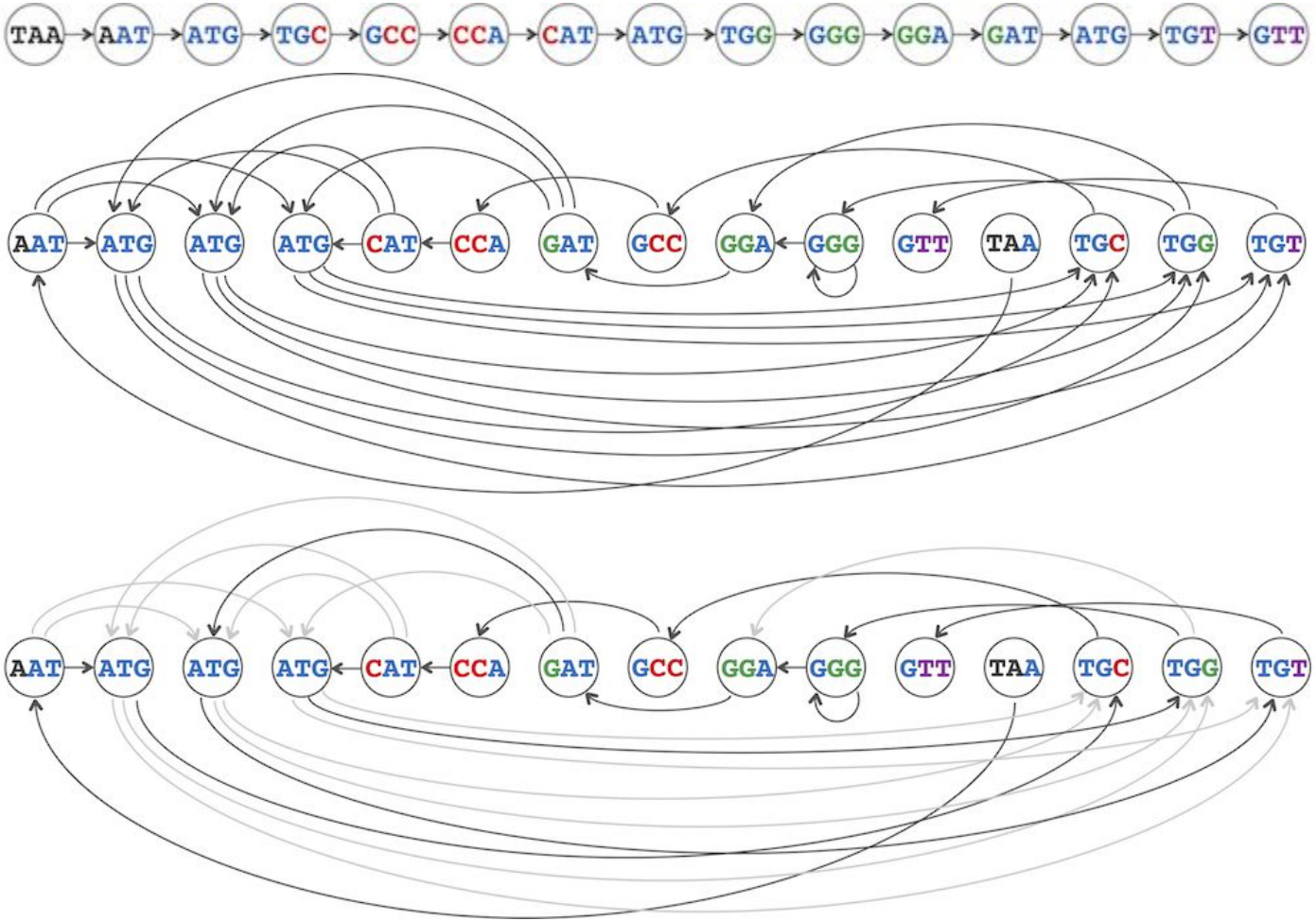
b is adjacent to *c* and *d*

c is adjacent to *a*

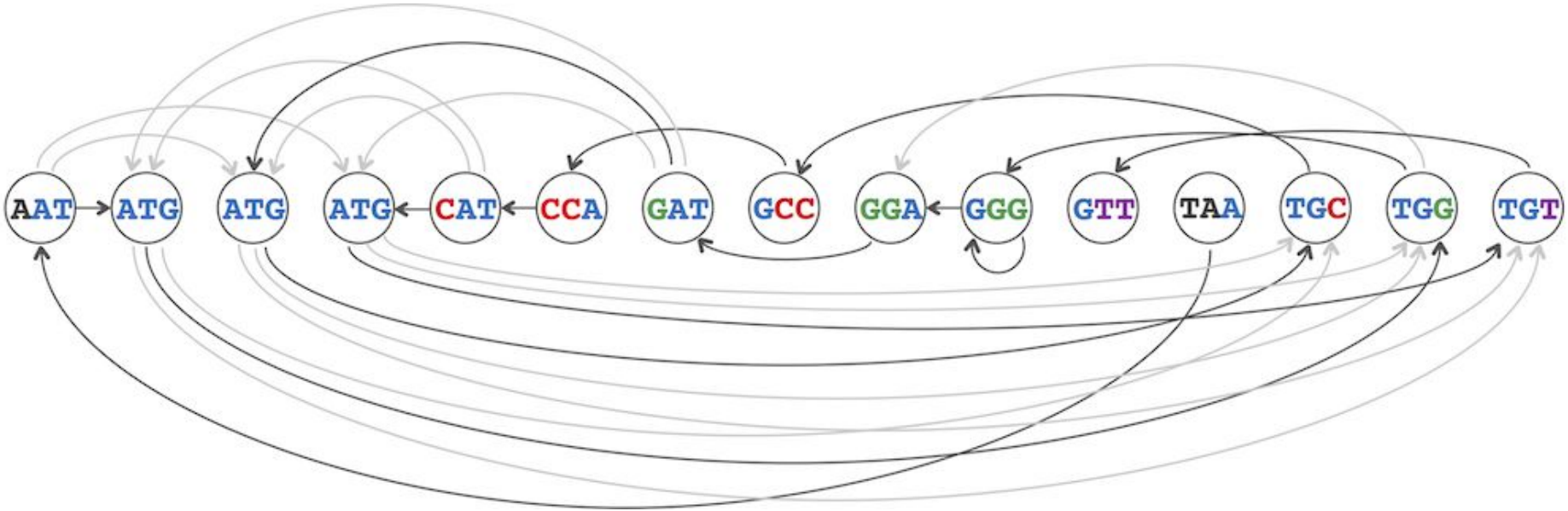
d is adjacent to *a*

e is adjacent to *b*, *c*, and *d*

Graphs


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

Hamiltonian path



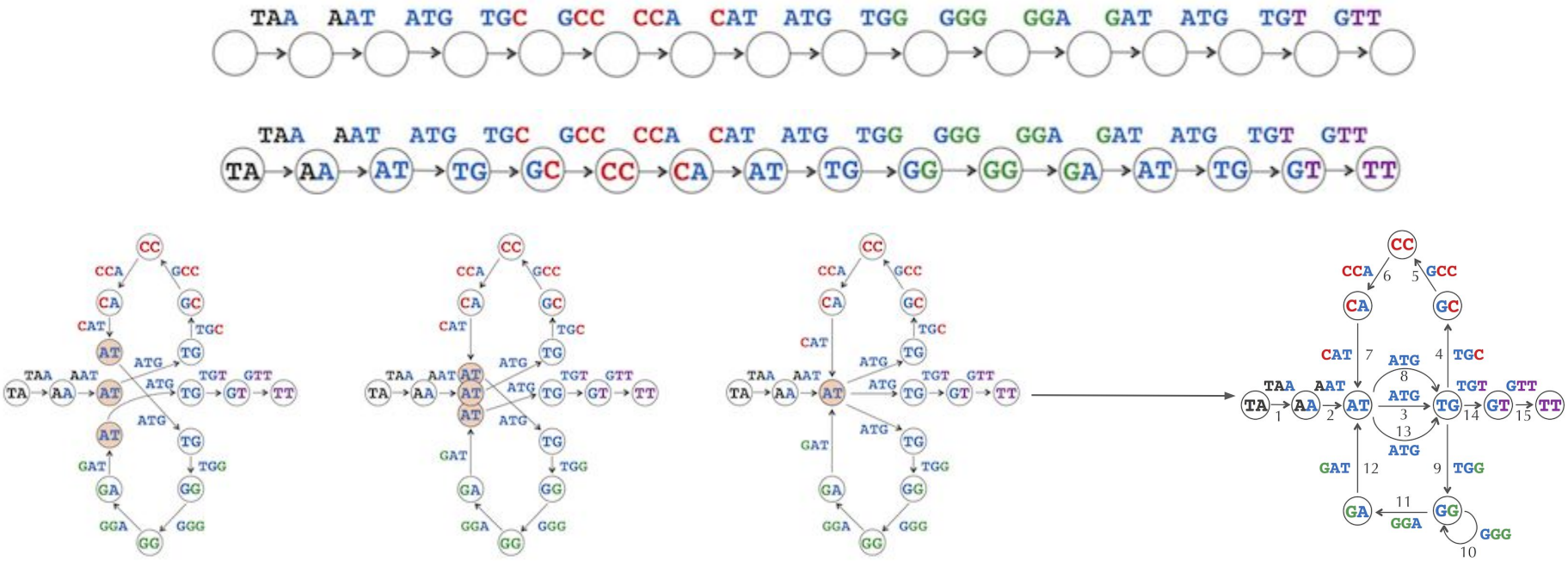
A path in a graph visiting every node once is called a **Hamiltonian path**

de Bruijn graph

Let's again represent the genome **TA**ATGCCATGGGATGTT as a sequence of its 3-mers:

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

This time, instead of assigning these 3-mers to *nodes*, we will assign them to *edges*, as shown in the figure below.



de Bruijn graph

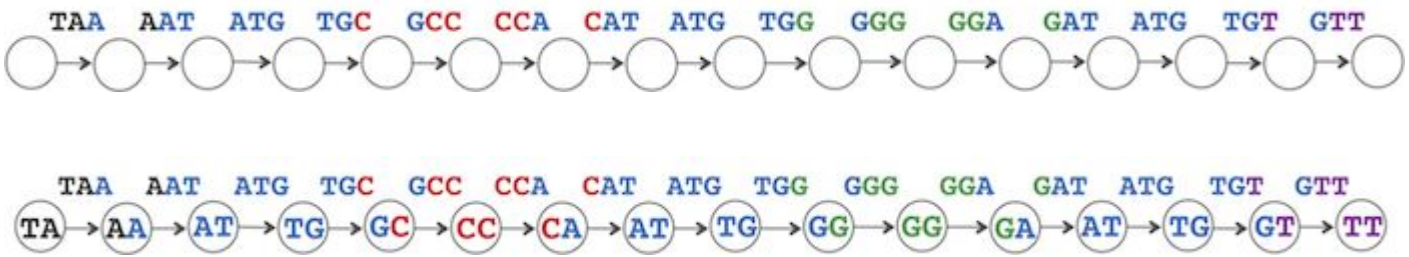
De Bruijn Graph from a String Problem: *Construct the de Bruijn graph of a string.*
Input: An integer *k* and a string *Text*.
Output: *DeBruijn*$_k$(*Text*).

Sample Input:

```
4
AAGATTCTCTAAGA
```

Sample Output:

```
AAG -> AGA,AGA
AGA -> GAT
ATT -> TTC
CTA -> TAA
CTC -> TCT
GAT -> ATT
TAA -> AAG
TCT -> CTA,CTC
TTC -> TCT
```



de Bruijn graph

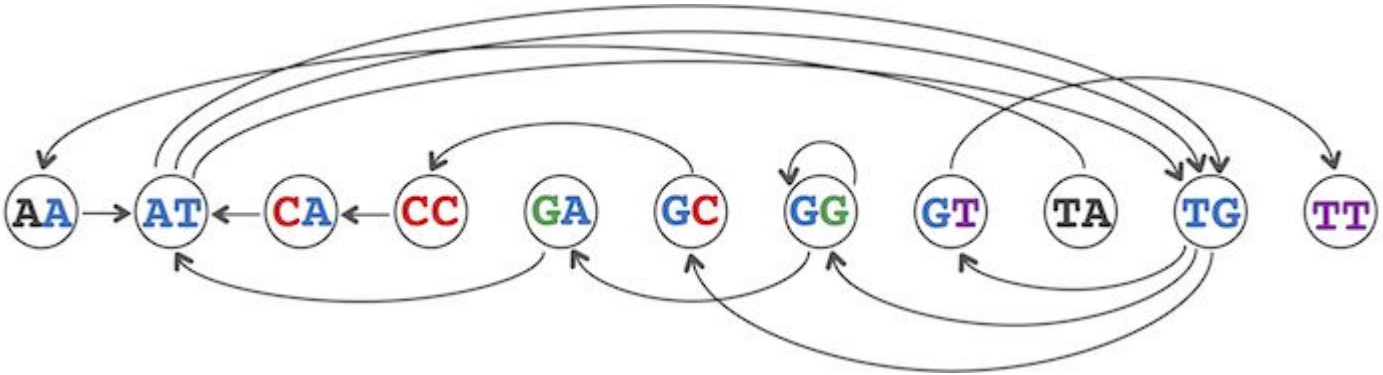
Given a collection of *k*-mers *Patterns*, the nodes of *DeBruijn*$_k$(*Patterns*) are all unique (*k*−1)-mers occurring as a prefix or suffix in *Patterns*. For example, say we are given the following collection of 3-mers:

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

Then the set of eleven *unique* 2-mers occurring as a prefix or suffix of 3-mers in this collection is as follows:

AA AT CA CC GA GC GG GT TA TG TT

For every *k*-mer in *Patterns*, we connect its prefix node to its suffix node by a directed edge in order to produce *DeBruijn*(*Patterns*). You can verify that this process produces the same de Bruijn graph that we have been working with (shown below).



de Bruijn graph

DeBruijn Graph from *k*-mers Problem: *Construct the de Bruijn graph from a set of k-mers.*

Input: A collection of *k*-mers *Patterns*.

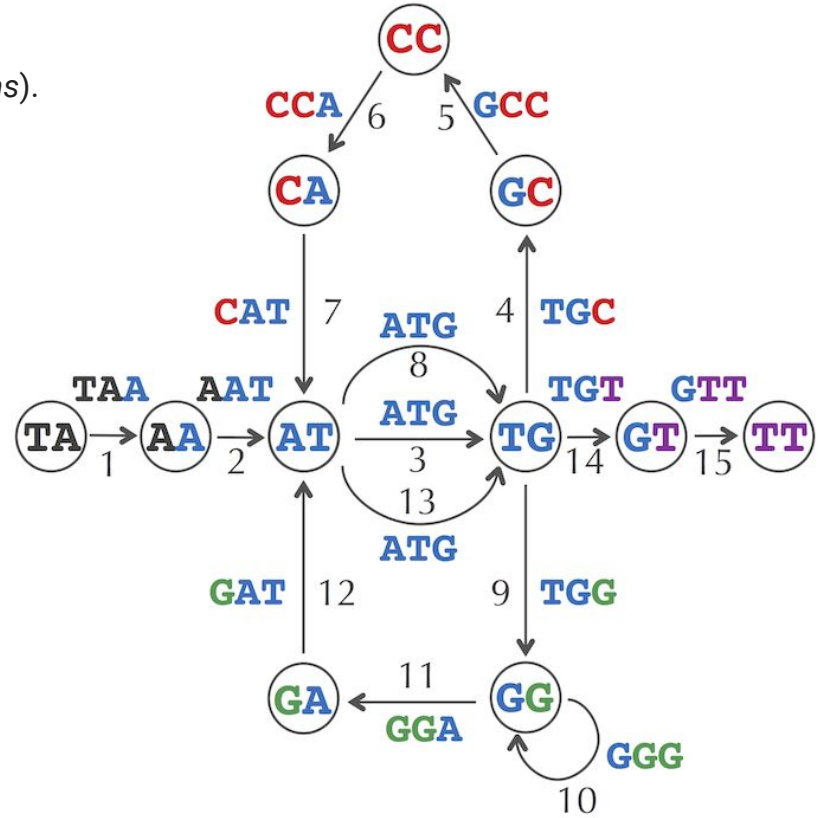
Output: The adjacency list of the de Bruijn graph *DeBruijn*(*Patterns*).

Sample Input:

```
GAGG
CAGG
GGGG
GGGA
CAGG
AGGG
GGAG
```

Sample Output:

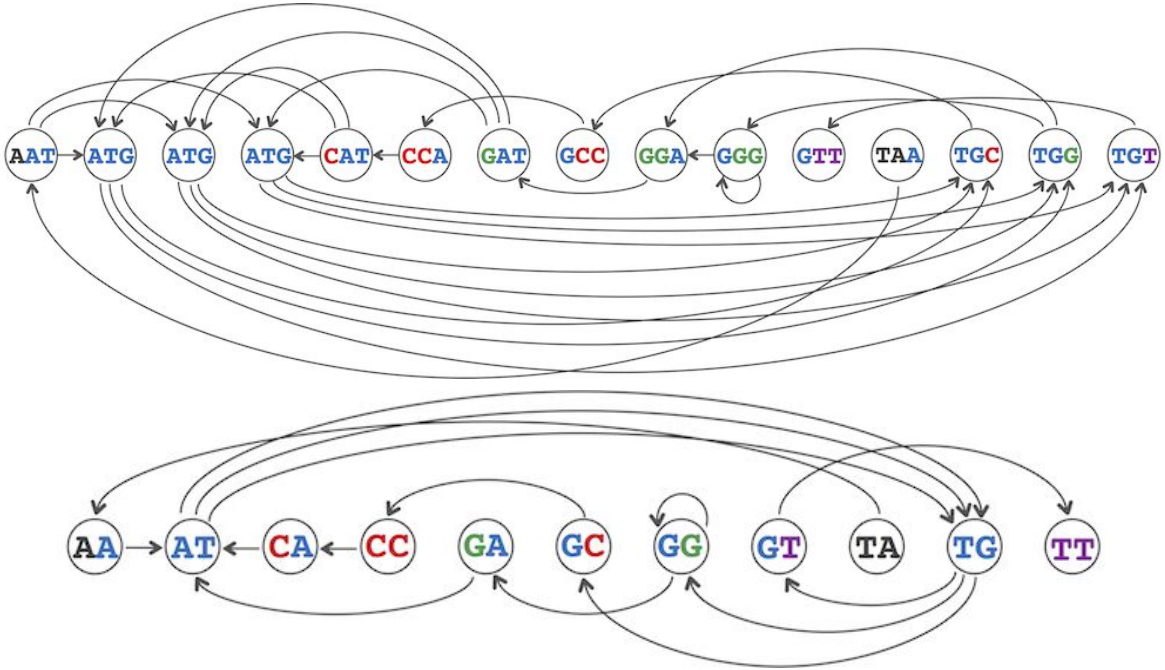
```
AGG -> GGG
CAG -> AGG,AGG
GAG -> AGG
GGA -> GAG
GGG -> GGA,GGG
```



Hamilton vs Euler

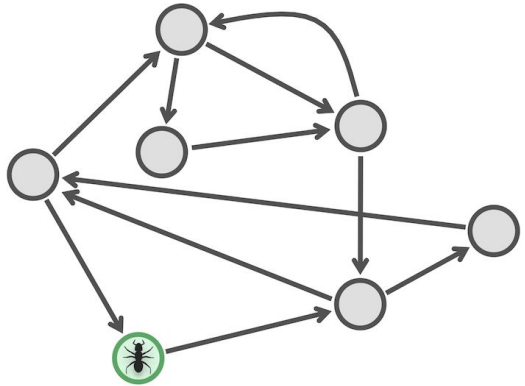
Finding a path in the de Bruijn graph that visits every *edge* exactly once is called an **Eulerian Path**

We now have two ways of solving the String Reconstruction Problem. We can either find a Hamiltonian path in the overlap graph (top) or find an Eulerian path in the de Bruijn graph (bottom).



The Euler cycle

We define the **indegree** and **outdegree** of a node *v* (denoted *in*(*v*) and *out*(*v*), respectively) as the number of edges leading into and out of *v*. A node *v* is **balanced** if *in*(*v*) = *out*(*v*), and a graph is **balanced** if all its nodes are balanced.

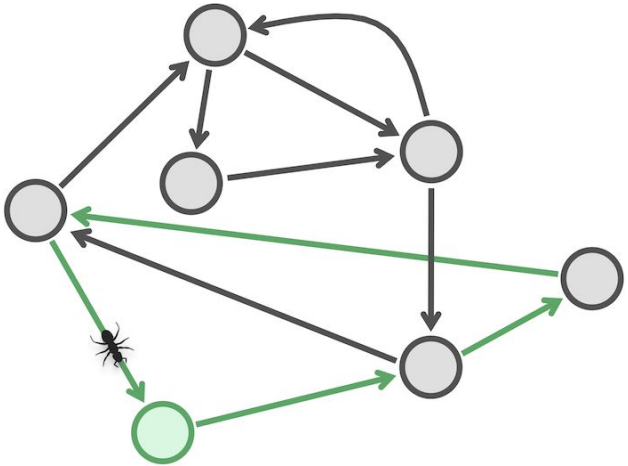


Let *Graph* be an arbitrary balanced and strongly connected directed graph. We place Leo at any node v_0 of *Graph* and let him randomly walk through the graph under the condition that he cannot traverse the same edge twice.

If Leo is incredibly lucky — or a genius — then he would traverse each edge exactly once and return back to v_0. However, odds are that Leo will “get stuck" somewhere before he can complete an Eulerian cycle, meaning that he reaches a node and finds no unused edges leaving that node.

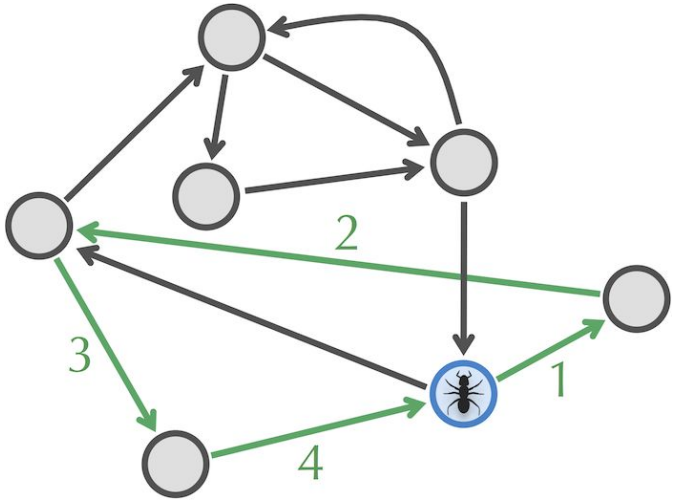
Where is Leo when he gets stuck? Can he get stuck in any node of the graph or only in some nodes?

The Euler cycle



It turns out that the only node where Leo can get stuck is the starting node v_0! The reason why is that *Graph* is balanced — if Leo walks into any node other than v_0 (through an incoming edge), then he will always be able to escape via an unused outgoing edge. The only exception to this rule is the starting node v_0, since Leo used up one of the outgoing edges of v_0 on his first move. Now, because Leo has returned to v_0, the result of his walk was a cycle, which we call $Cycle_0$ (see the figure above).

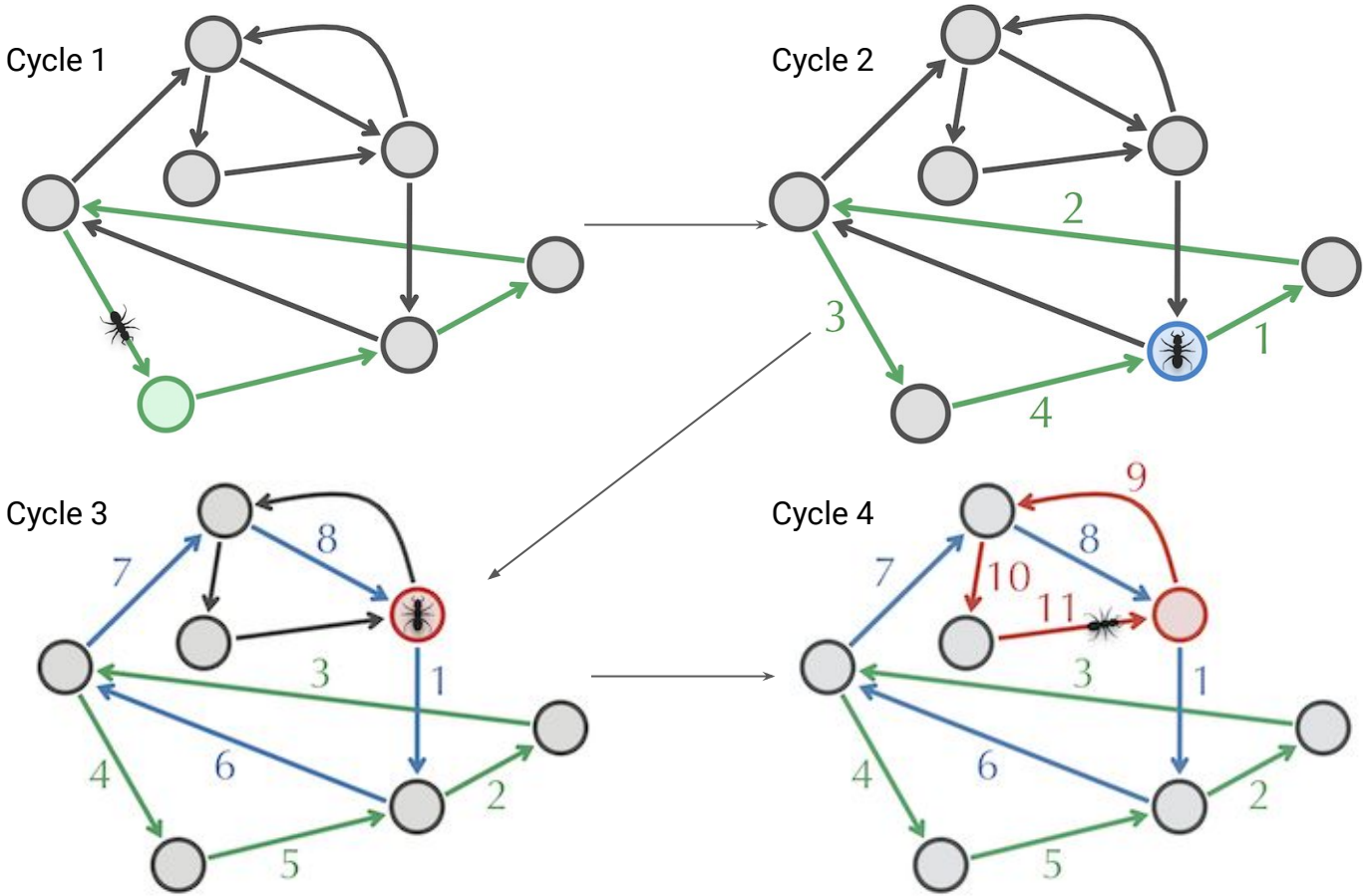
The Euler cycle



If *Cycle*$_0$ is Eulerian, we are finished. Otherwise, because *Graph* is strongly connected, some node on *Cycle*$_0$ must have unused edges entering it and leaving it (why?). Naming this node v_1 , we ask Leo to start at v_1 instead of v_0 and traverse *Cycle*$_0$ (thus returning to v_1), as shown in the figure above.

He will eventually get stuck at v_1, the node where he started. However, now there are unused edges starting at this node, and so he can continue walking from v_1, using a new edge each time. The same argument as the one that we used above implies that Leo must eventually get stuck at v_1. The result of Leo's walk is a new cycle, *Cycle*$_1$ (see the figure above), that is larger than *Cycle*$_0$.

The Euler cycle



The Euler cycle

```
EulerianCycle(Graph)
       form a cycle Cycle by randomly walking in Graph (don't visit the same edge twice!)
       while there are unexplored edges in Graph
           select a node newStart in Cycle with still unexplored edges
           form Cycle' by traversing Cycle (starting at newStart) and then randomly walking
           Cycle ← Cycle'
       return Cycle
```

Code Challenge: Solve the Eulerian Cycle Problem.
Input: The adjacency list of an Eulerian directed graph.
Output: An Eulerian cycle in this graph.

Sample Input:

```
0 -> 3
1 -> 0
2 -> 1,6
3 -> 2
4 -> 2
5 -> 4
6 -> 5,8
7 -> 9
8 -> 7
9 -> 6
```

Sample Output:

```
6->8->7->9->6->5->4->2->1->0->3->2->6
```

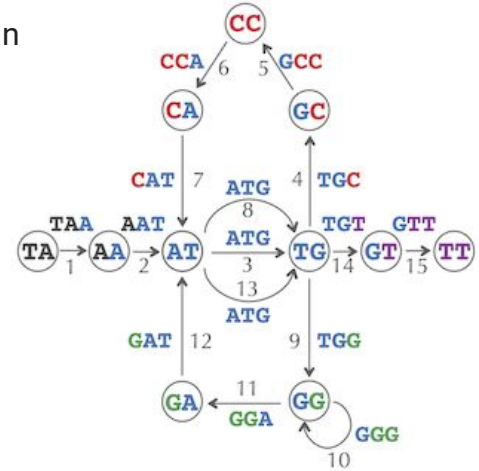
The Euler path

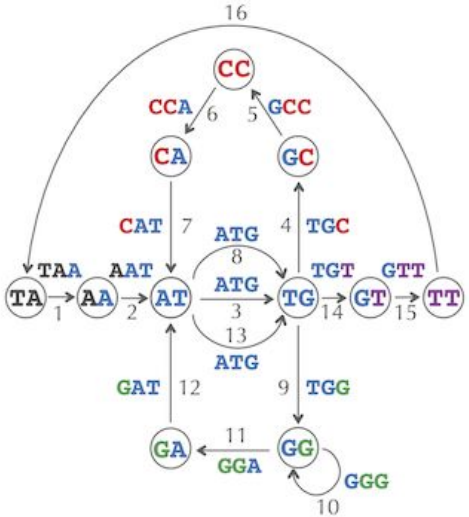
Consider the de Bruijn graph on the left in the figure below, which we already know has an Eulerian path, but which does not have an Eulerian cycle because nodes **TA** and **TT** are not balanced. However, we can transform this Eulerian path into an Eulerian cycle by adding a single edge connecting **TT** and **TA**, as shown in the figure below.

Code Challenge: Solve the Eulerian Path Problem.

Input: The adjacency list of a directed graph that has an Eulerian path.

Output: An Eulerian path in this graph.





Sample Input:

```
0 -> 2
1 -> 3
2 -> 1
3 -> 0,4
6 -> 3,7
7 -> 8
8 -> 9
9 -> 6
```

Sample Output:

```
6->7->8->9->6->3->0->2->1->3->4
```