Efficient Sequence Assembly

Sequencing by Hybridization (SBH)

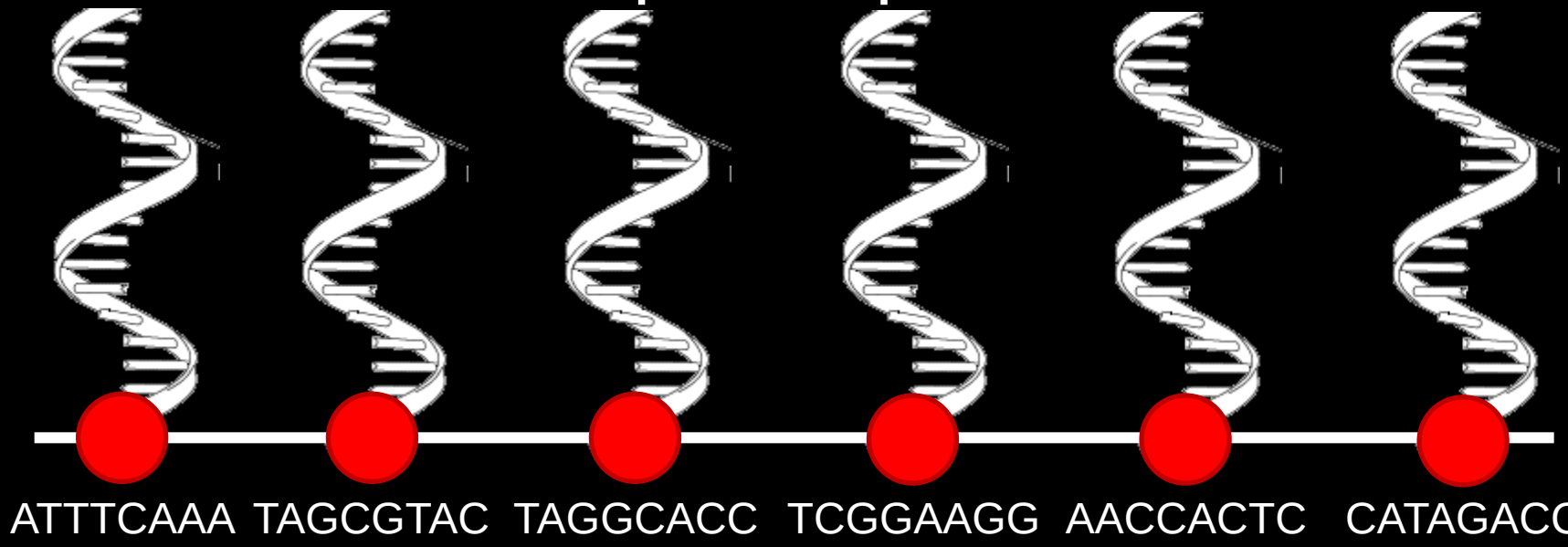
- Big picture:
  - You are getting your data from a DNA microarray
- DNA microarrays work in a very specific way:

1) A DNA sample is broken up into small single stranded pieces (~30 bases)
2) fluorescent dyes are attached to each piece

- These are floating around loose in solution
- If you shine a light with a specific frequency, these DNA will fluoresce

Sequencing by Hybridization phase 2

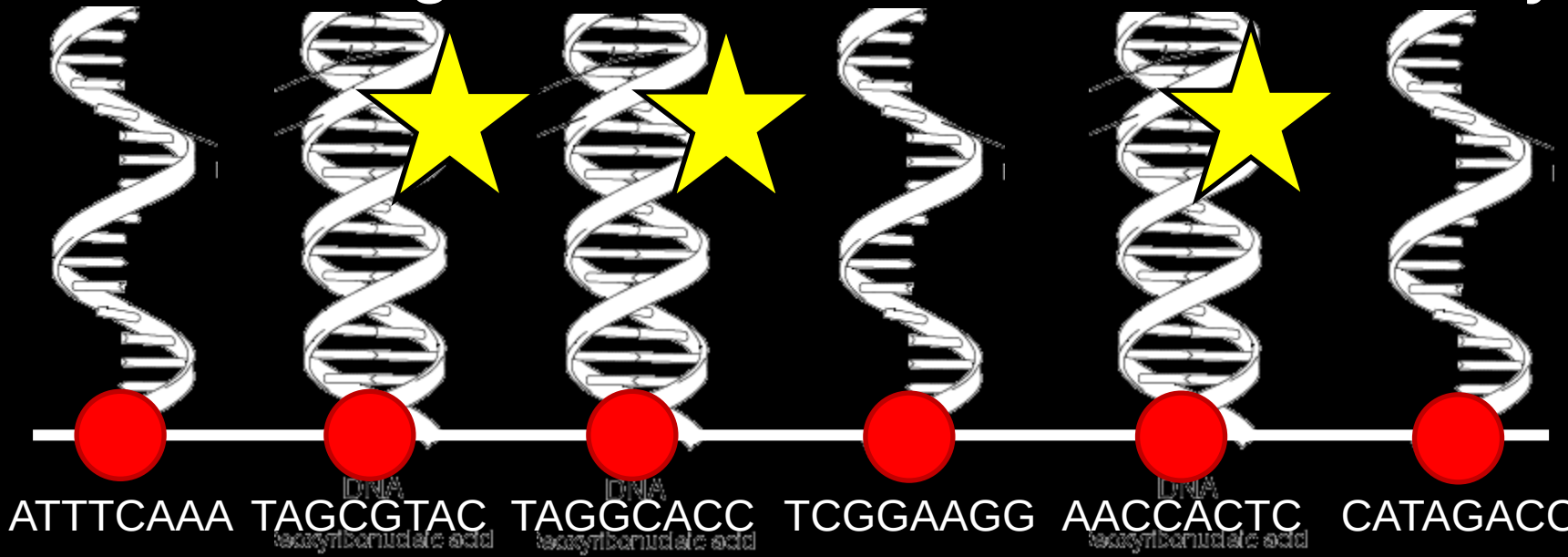
- Wash the loose pieces over a DNA microarray, which has a number of attached single stranded DNA in specific places



- The DNA microarray is a piece of glass with separated regions where these DNA are attached. The sequence in each region is known.

Sequencing by Hybridization phase 3

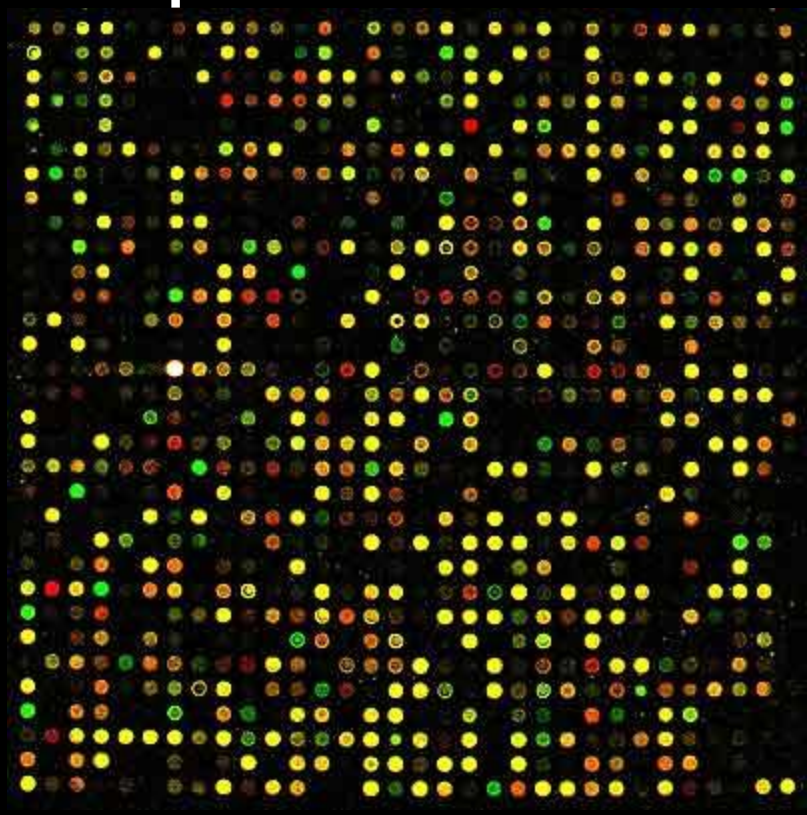
- The washing effect means that all the DNA pieces have a chance to touch all the DNA fixed to the glass. The rest are washed away.



- DNAs with complementing bases will stick to matching partners on the glass.
- Thus, some parts of the glass will fluoresce

Sequencing by Hybridization phase 4

- Scanning the chip with a digital camera indicates which regions of the chip the DNAs are bound in.
- Since we know the sequences of the DNAs we attached in each region, we know the sequences of the DNAs that attached to them
- These are reads.



Some minor caveats

- Sequencing by Hybridization is not perfect.
  - DNA that is not perfectly complementary can bind to regions of the DNA array, thereby incorrectly indicating the sequence
  - This is called cross-hybridization
    - We will assume here and in Project 1 that cross hybridization does not happen
- For reads of length n, DNA arrays do not generally have all 4^n possible fixed DNAs
  - We will assume that they do, for our problem size
- These simplifying assumptions do make the problem unrealistic, but they make it manageable for this course and Project 1

Lets get an idea of the data we have

- Sequence Length: 5000-15000 bases
- Reads: 30 bases
- Coverage: 50-100x coverage

  ATATATGGGCCACGATGTACGAGCCACTCACACA

- If you had a read starting at every base, then you would only have 30x coverage.
  - Two or more reads start at every base.
- Virii are short, so they can be sequenced to 300x-500x coverage easily, and new, faster technologies will have greater coverage for all.

Assembly is not the problem of finding the one read that will complete your contig.

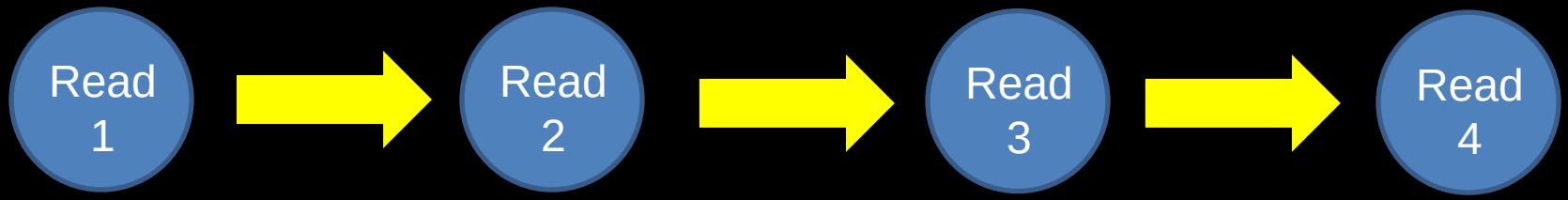
Simplistic Path assembly

- Pick one read A
- Look through all the other reads
  - Find a read B with a 15 base prefix that is identical to the 15 base suffix of A.
  - i.e. suffix(A, 15) == prefix(B,15)
- Look through all the other reads
  - Find a read C with a 15 base prefix that is identical to the 15 base suffix of B.
  - i.e. suffix(B,15) == prefix(C,15)
- Continue..

Why is simplistic path assembly bad?

- Because for each read, you have to look at every other read.
- 15000 bases covered 100 times by 30 base reads is 60,000 reads.
  - For each read, you must check every other read:
  - This is 3.6 billion checks.
  - If each check is 1 millisecond, we're talking 1000 hours
- This is really slow. We can do way better.

Eulerian Path Assembly



- The problem with simple assembly is that we are forced to start on one end and go through all the reads each time we want to extend the contig.
- What we really need is to go through all the reads once, and have it build the contig as we go.
- Eulerian Path Assembly can do this.
- First we need to learn a little graph theory.

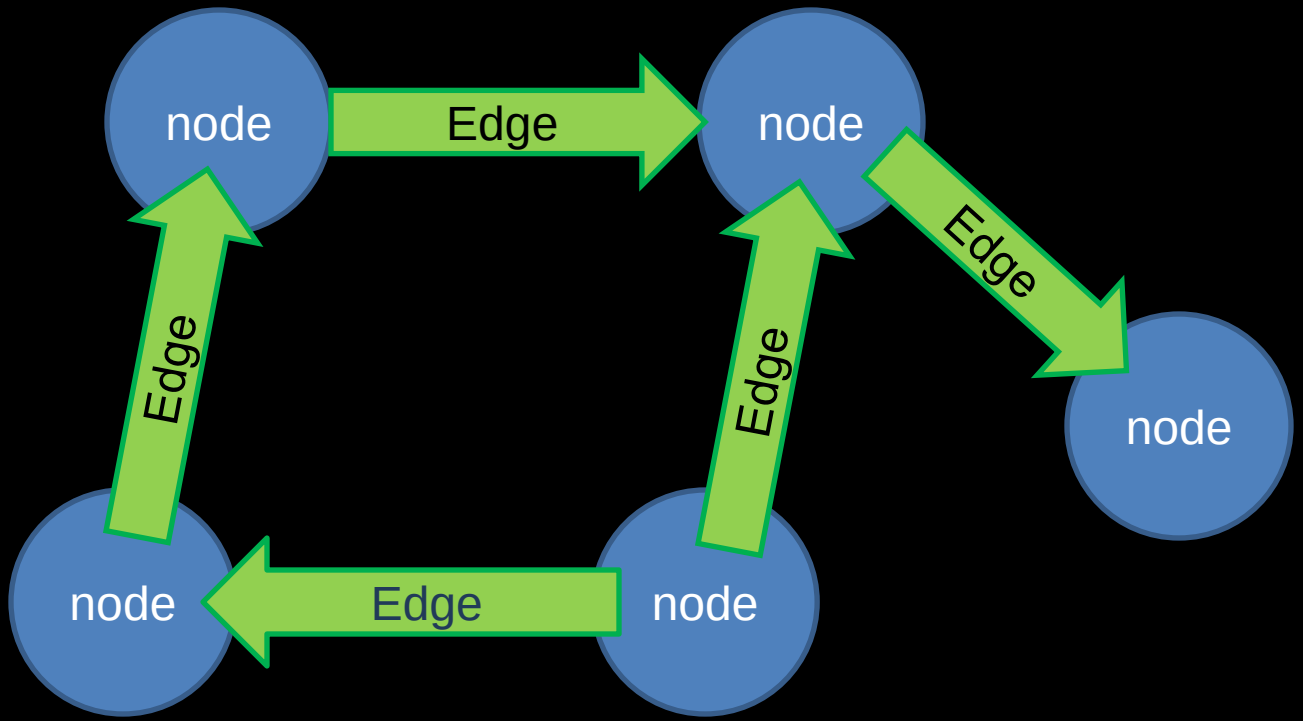
Terminology

...CACGATGTACGAGCCACTCA
GAGCCACTCAATCTATTTGC...

Overlapping reads

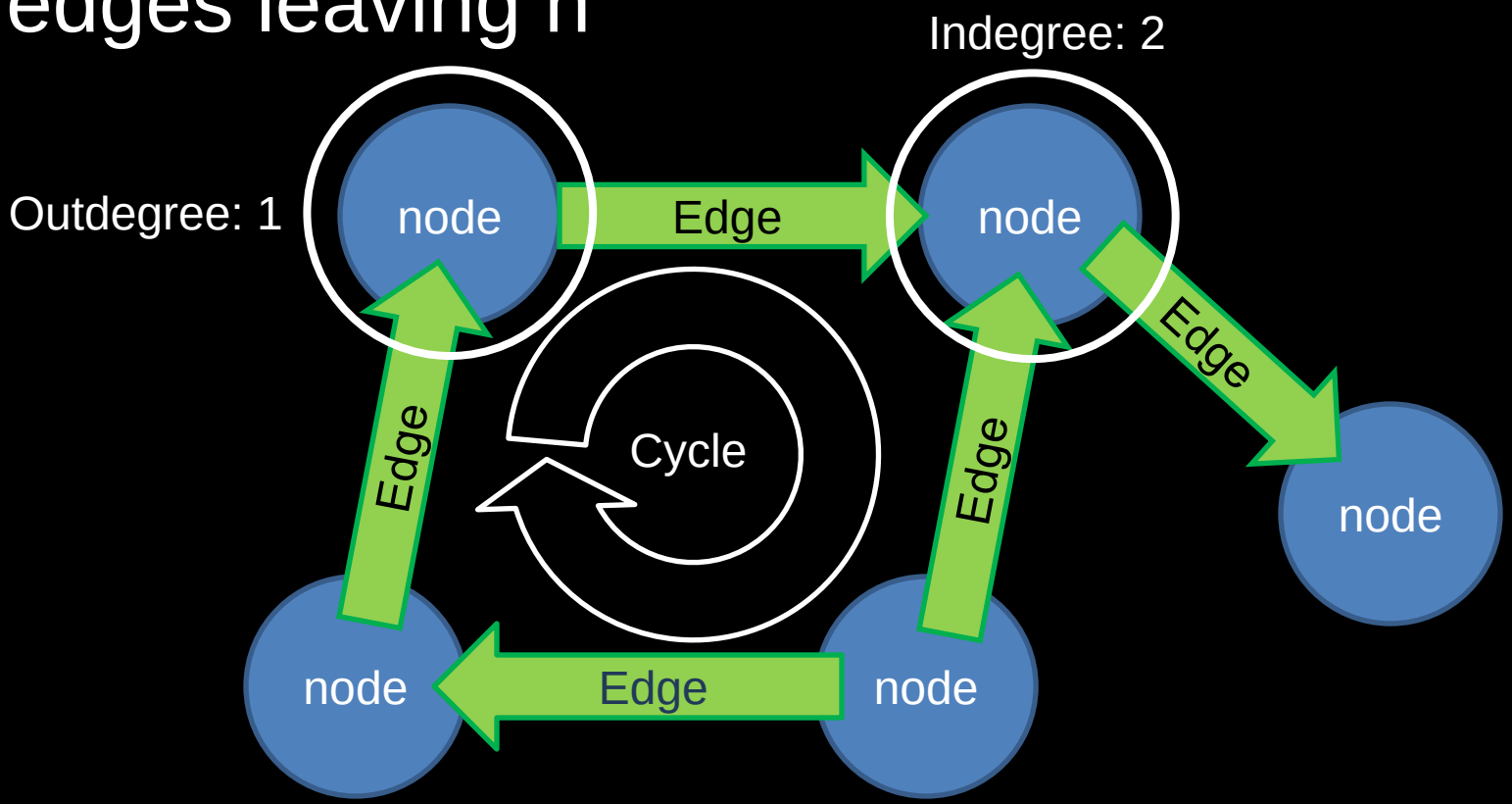
What is a graph?

- Graphs look like this:
  - Nodes represent things
  - Edges directionally connect two nodes
- Edges and nodes can stand for many things



Some things to know about graphs

- The *indegree* of a node n is the number of edges going into n
- The outdegree of a node n is the number of edges leaving n



Using Graphs for Sequence Assembly

- Each node will stand for either a prefix or a suffix.
  - Prefixes and suffixes are the same length, so each node will be used to represent both.

- Each edge will stand for a read, which connects a prefix to a suffix

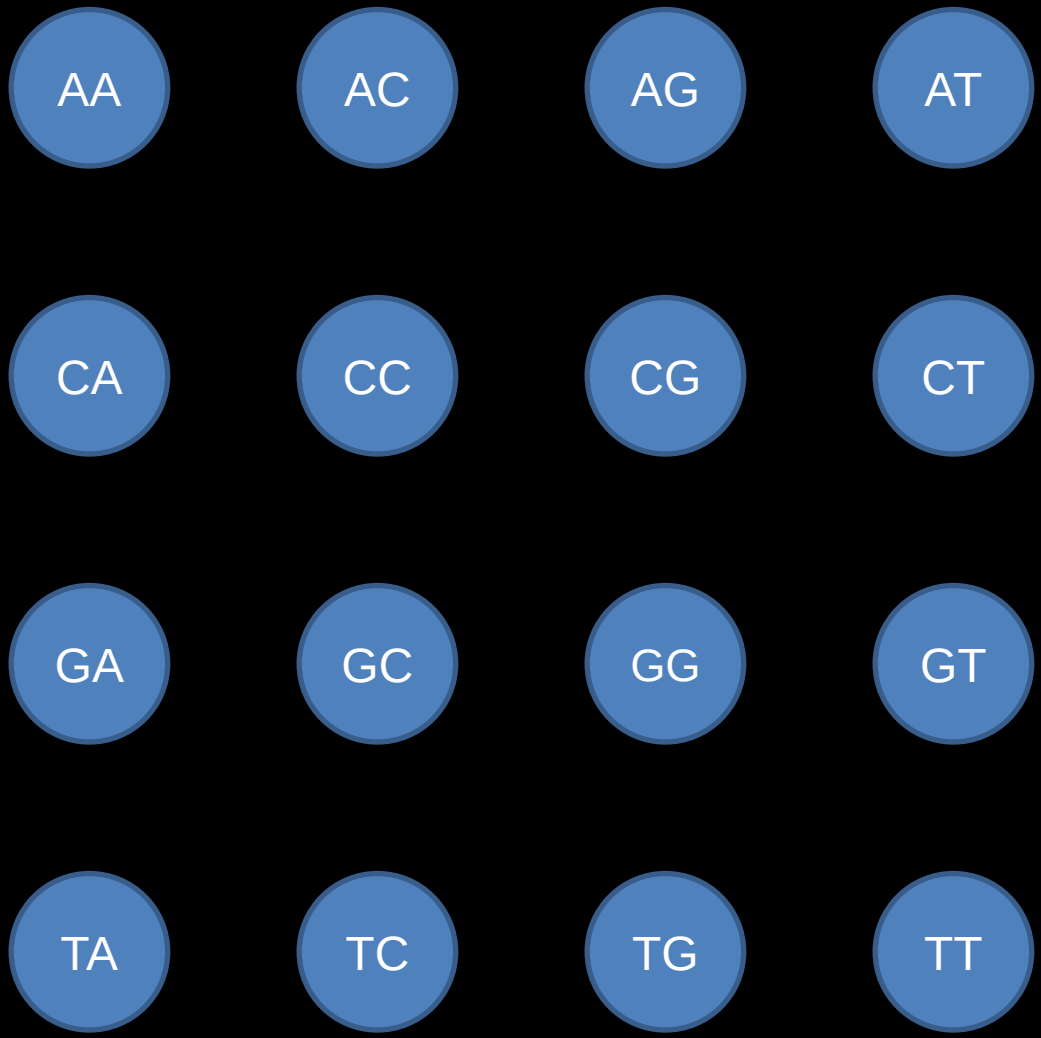
A really simple example

- For this example:
  - Reads are 3 bases
  - Prefixes are 2 bases, suffixes are 2 bases
  - (they overlap, and that's okay)
- Contig: {ATGGCGTGCA}
- Reads:

  {ATG, TGG, TGC, GTG, GGC, GCA, GCG, CGT}
- Prefixes and suffixes:

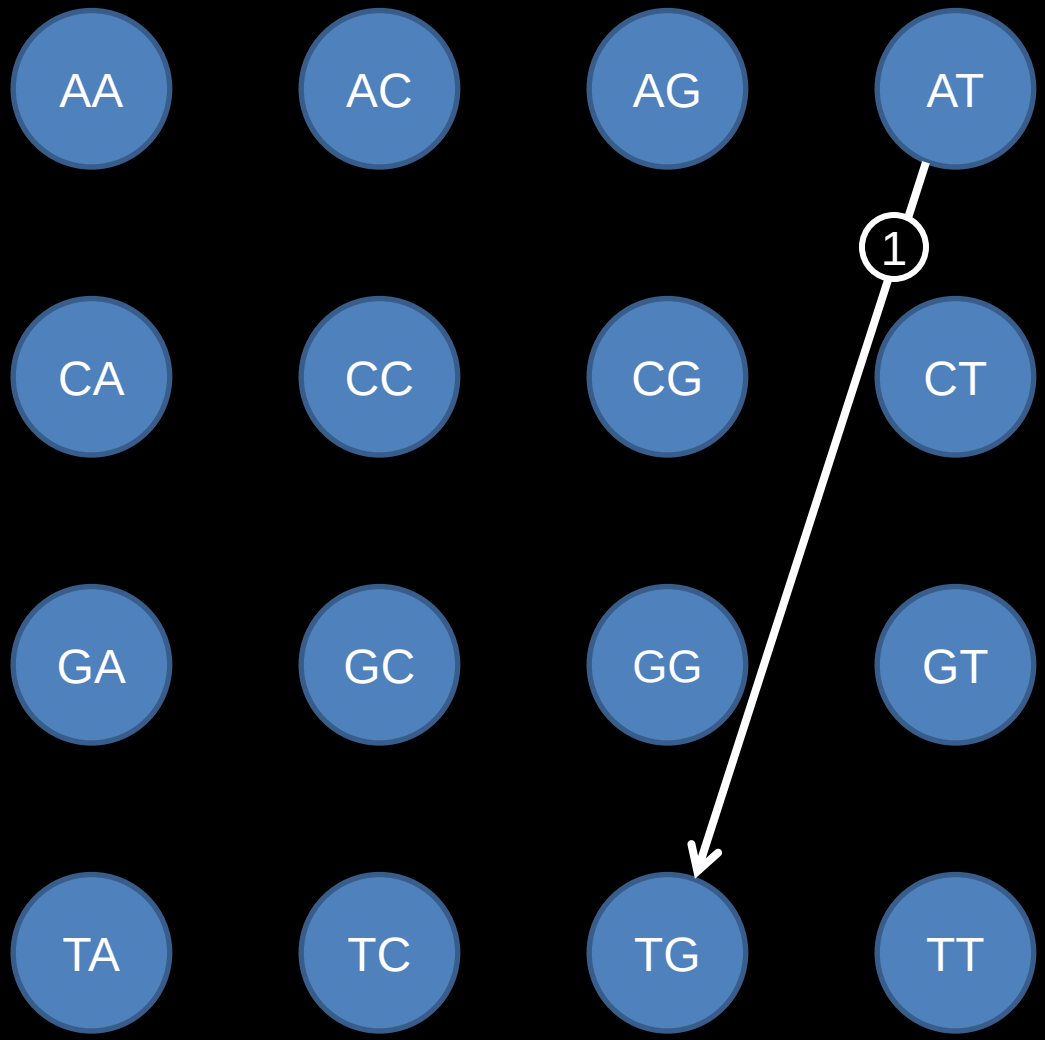
  {AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, TT}
- This is all 4x4 combinations of 2-letter prefixes and suffixes.

Step1: Prefixes and Suffixes are nodes



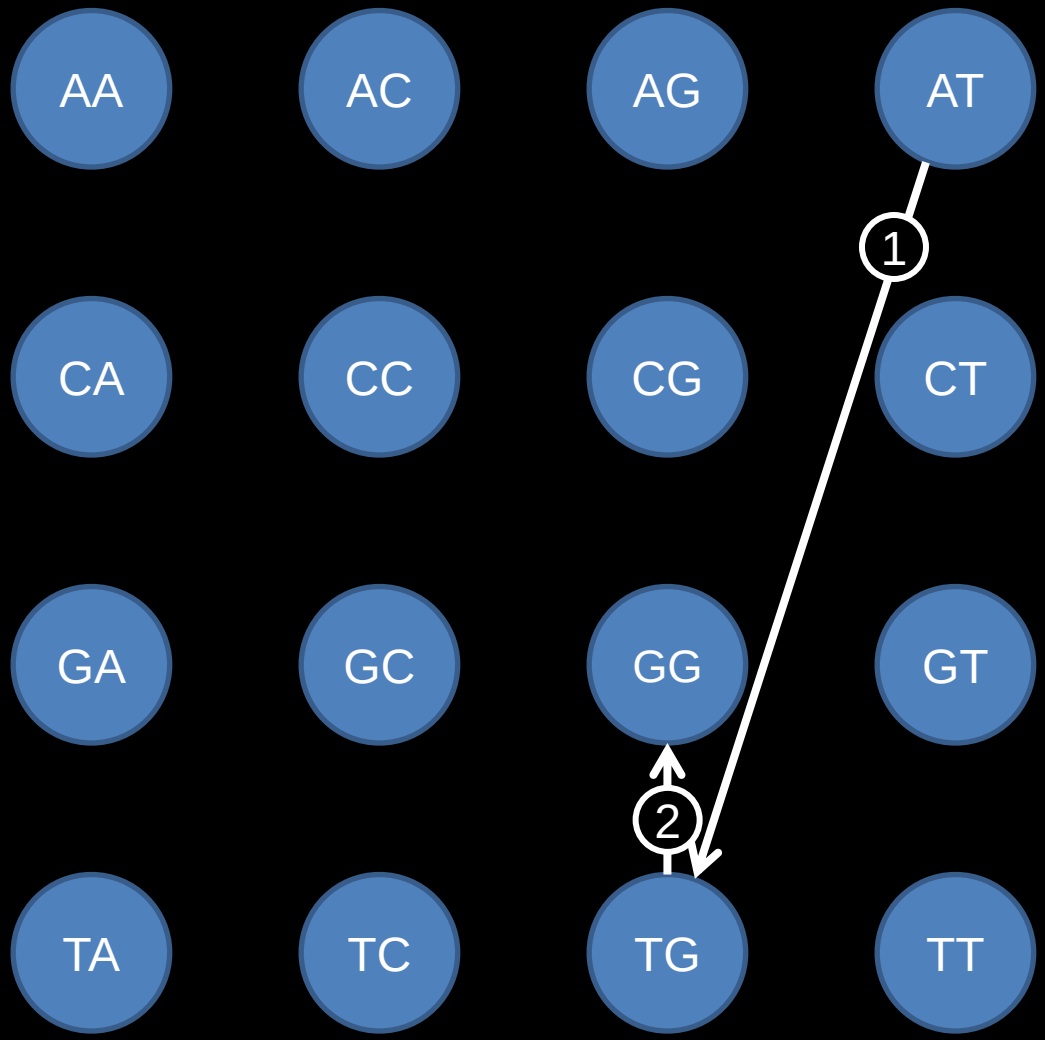
Reads are Edges

1	~~ATG~~
2	TGG
3	TGC
4	GTG
5	GGC
6	GCA
7	GCG
8	CGT



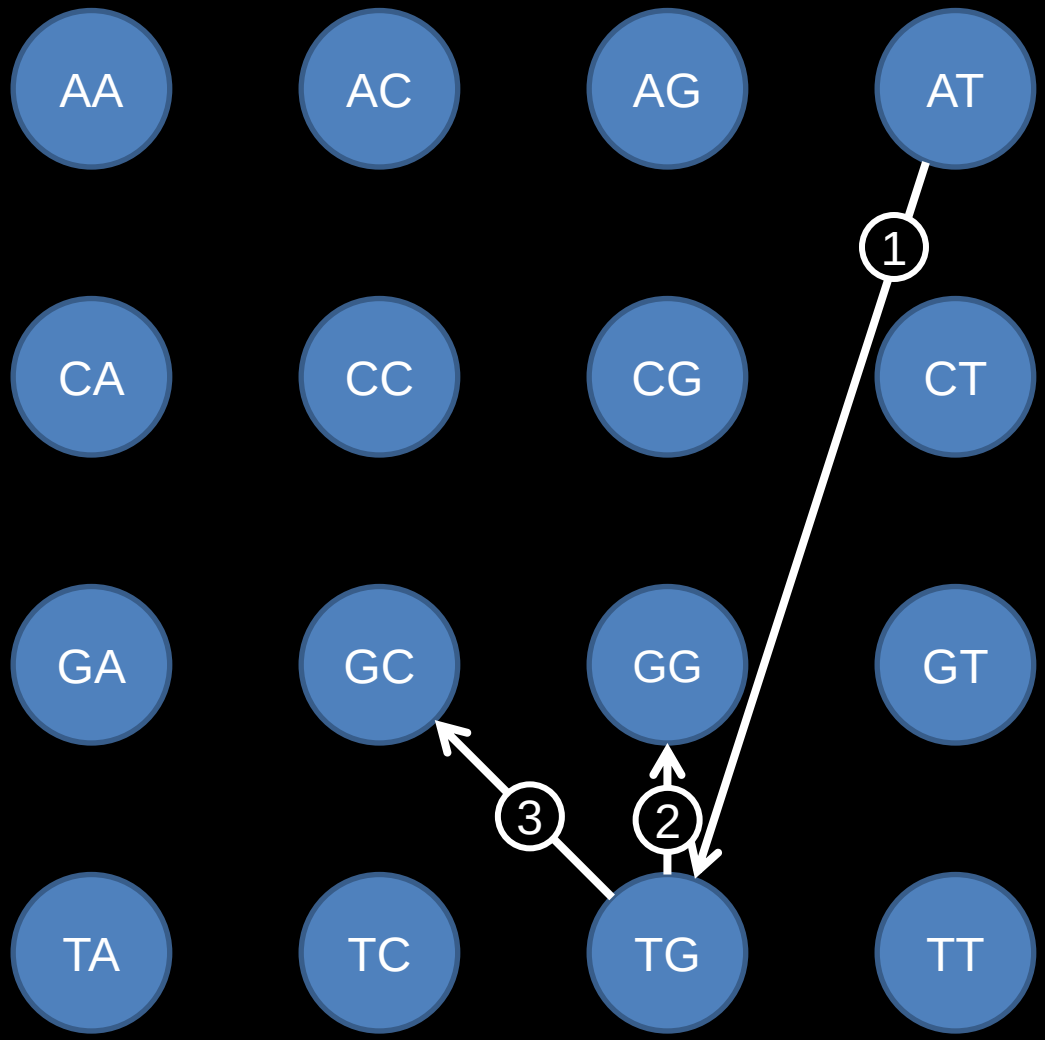
Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	TGC
4	GTG
5	GGC
6	GCA
7	GCG
8	CGT



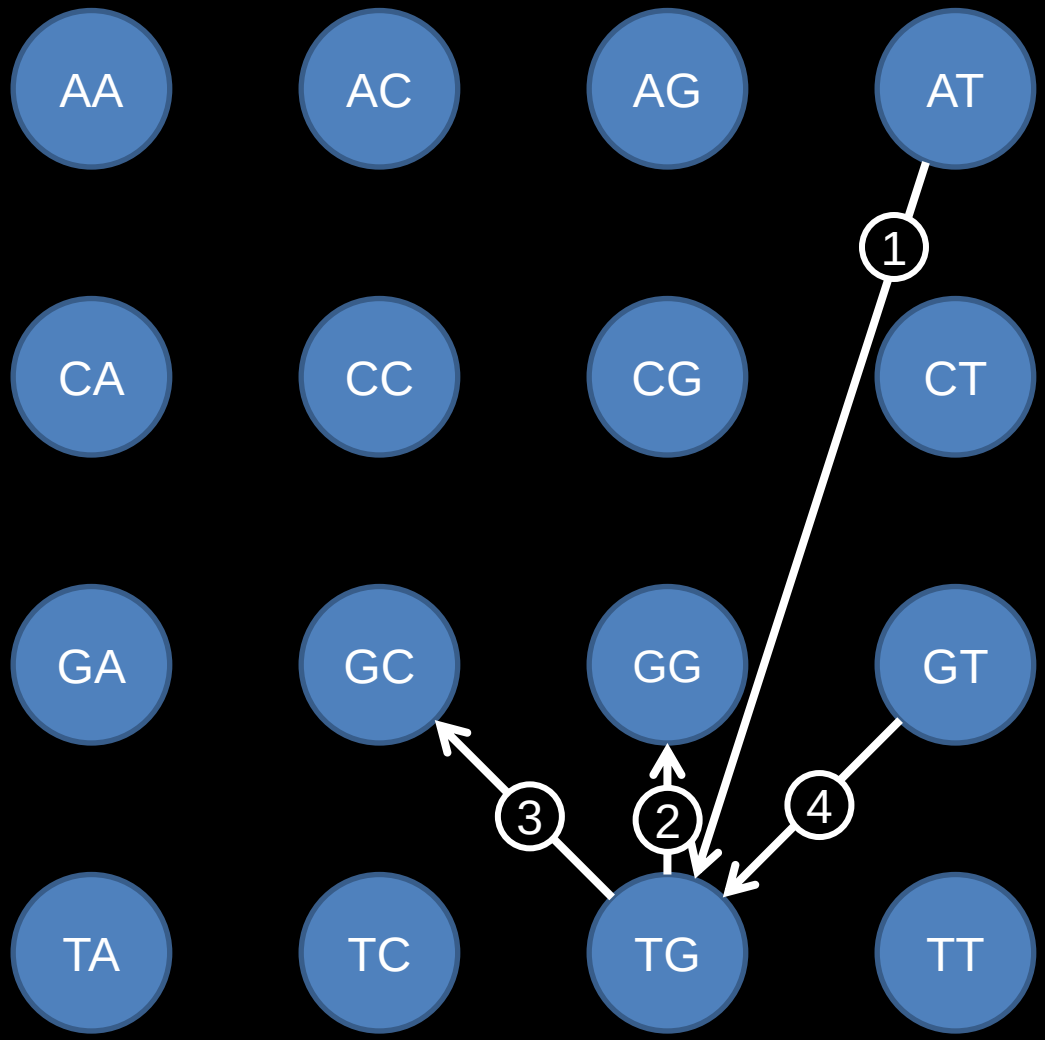
Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	~~TGC~~
4	GTG
5	GGC
6	GCA
7	GCG
8	CGT



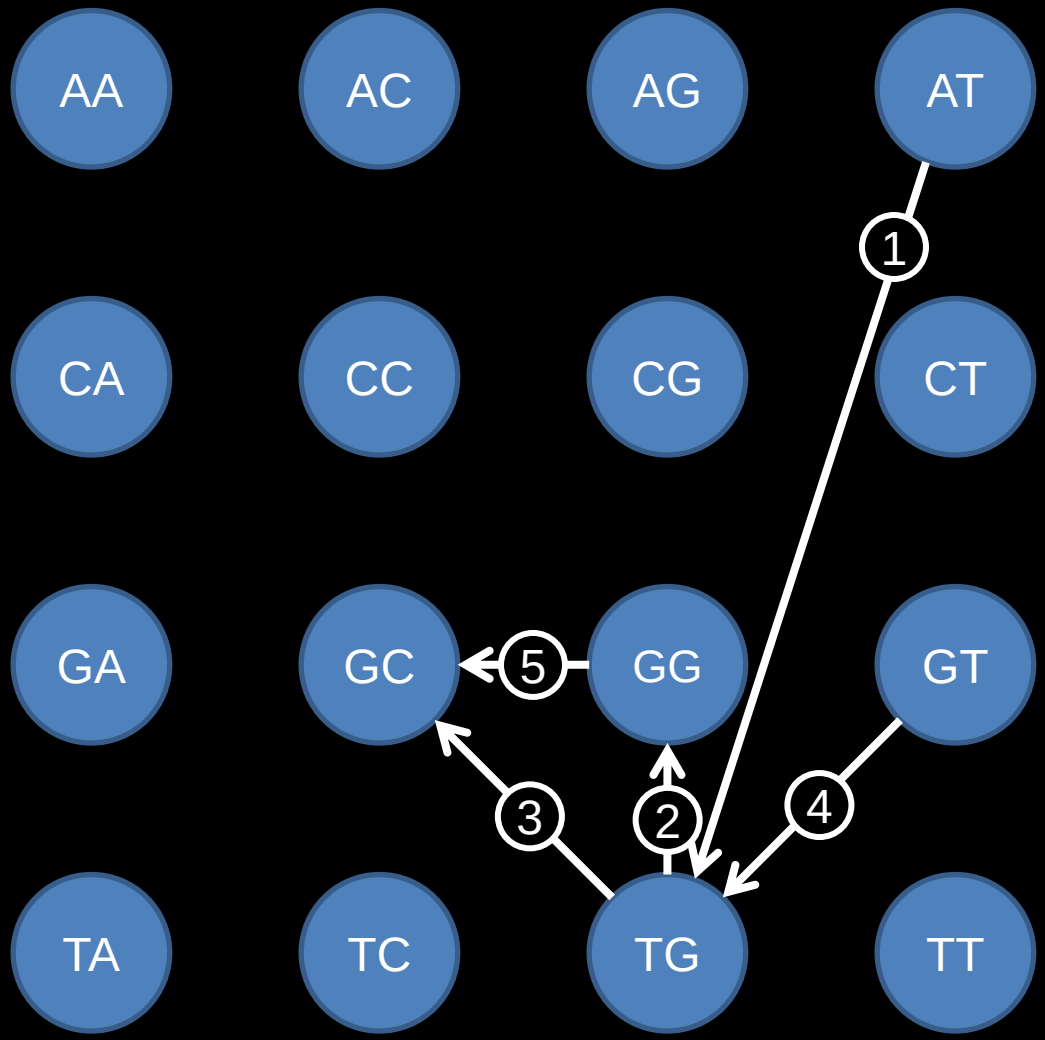
Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	~~TGC~~
4	~~GTG~~
5	GGC
6	GCA
7	GCG
8	CGT



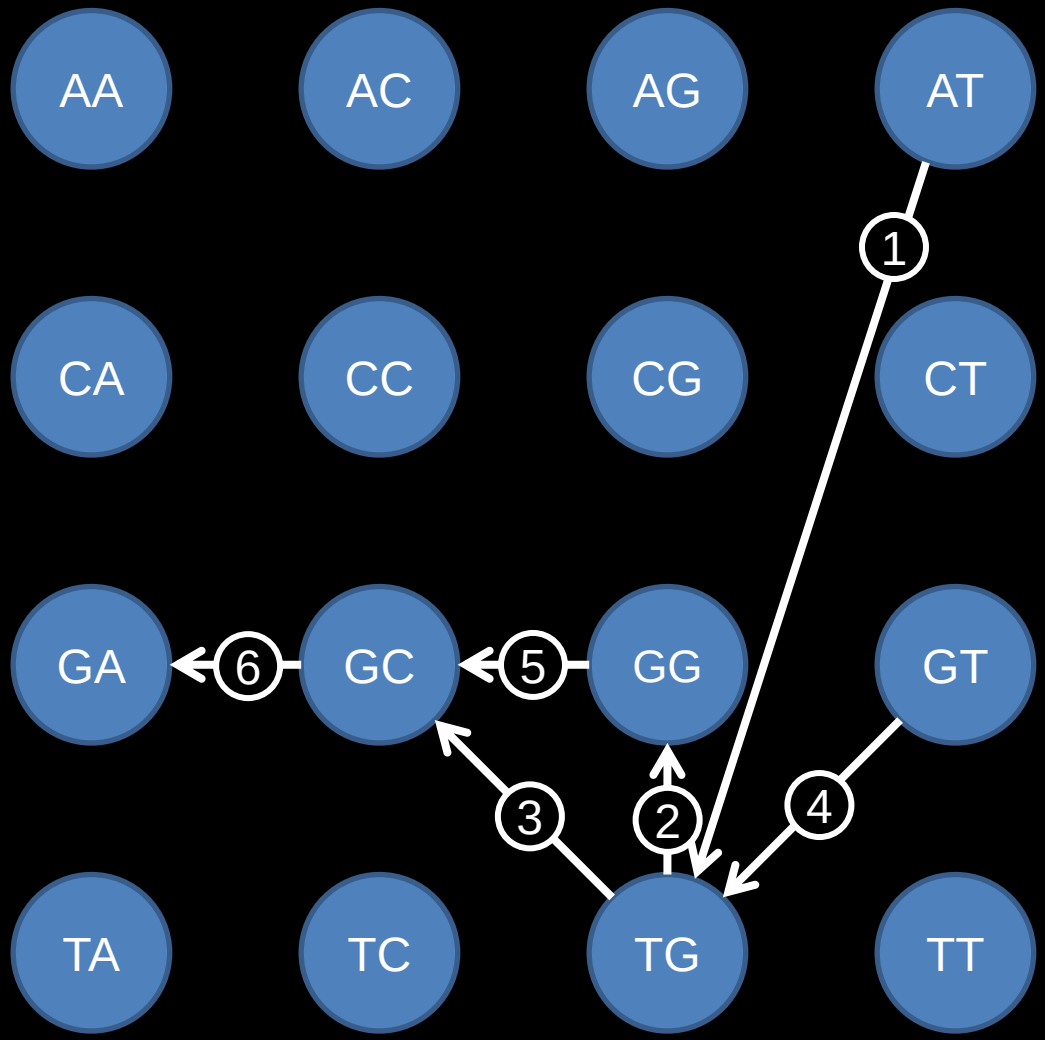
Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	~~TGC~~
4	~~GTG~~
5	~~GGC~~
6	GCA
7	GCG
8	CGT



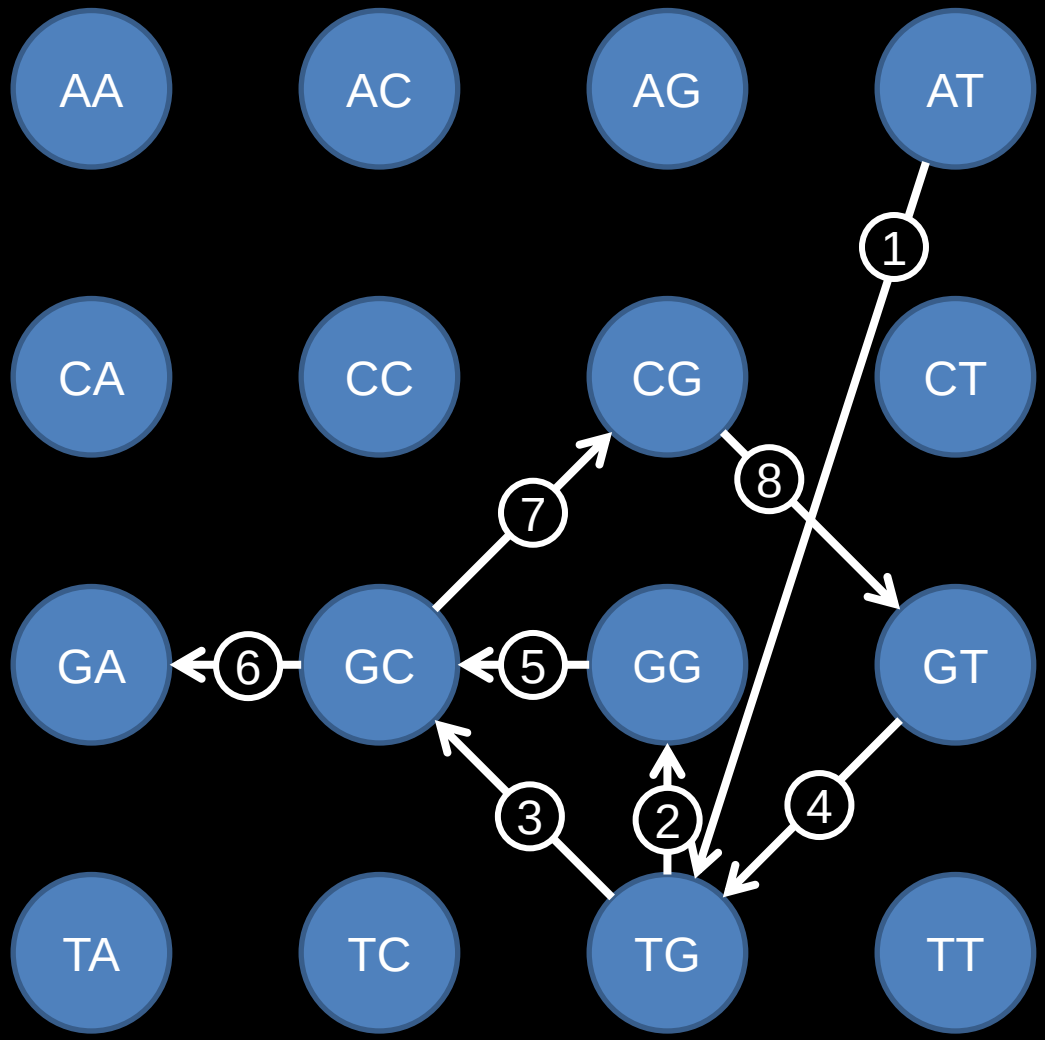
Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	~~TGC~~
4	~~GTG~~
5	~~GGC~~
6	~~GCA~~
7	GCG
8	CGT

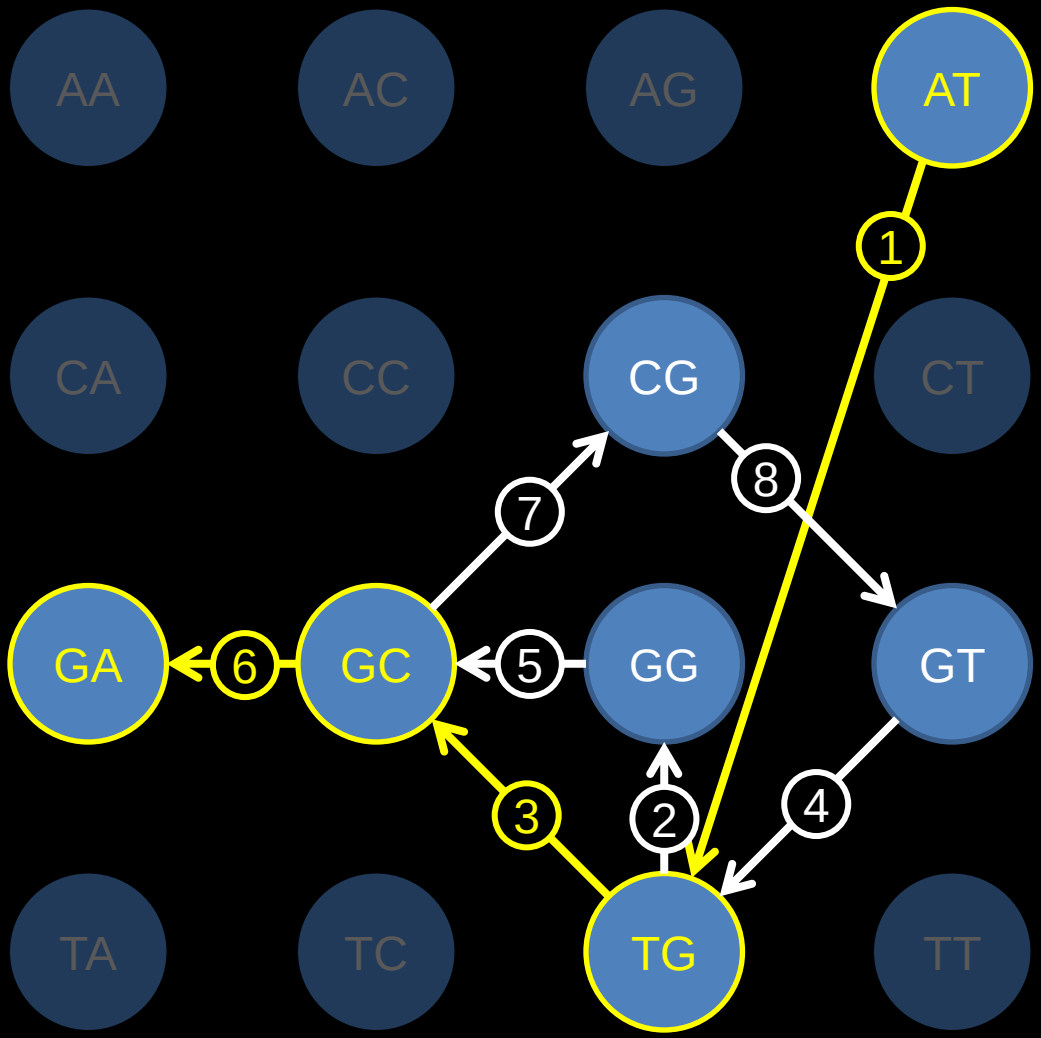


Reads are Edges

1	~~ATG~~
2	~~TGG~~
3	~~TGC~~
4	~~GTG~~
5	~~GGC~~
6	~~GCA~~
7	~~GCG~~
8	~~CGT~~



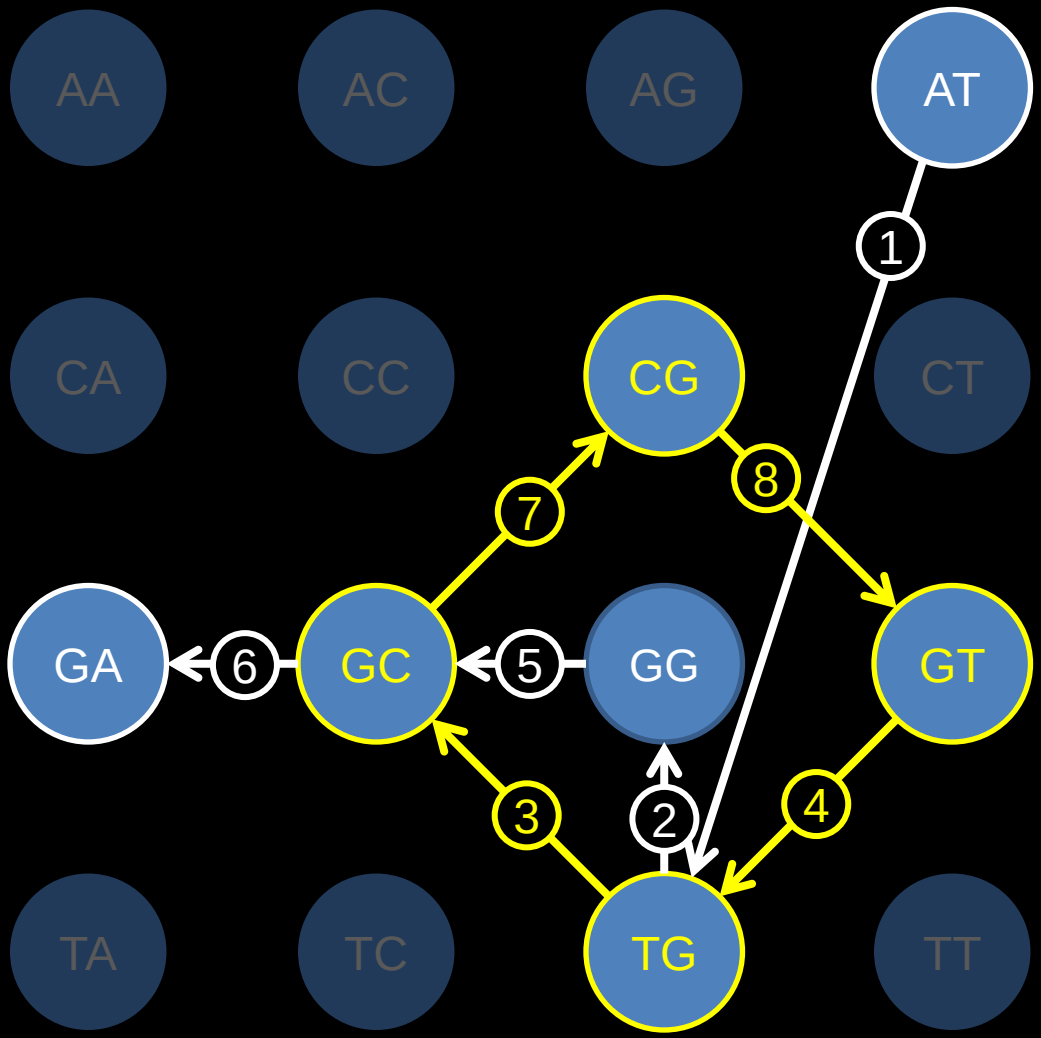
How do we interpret this?



- Each node represents a prefix or suffix.
- Start with an edge with outdegree > indegree
- Assemble the contigs in any edge order:

  1 3 6
- You cannot reuse edges!
  - Each edge is a read
  - Frequently there will be identical reads: multiple edges between the same nodes.

Sometimes you find cycles:



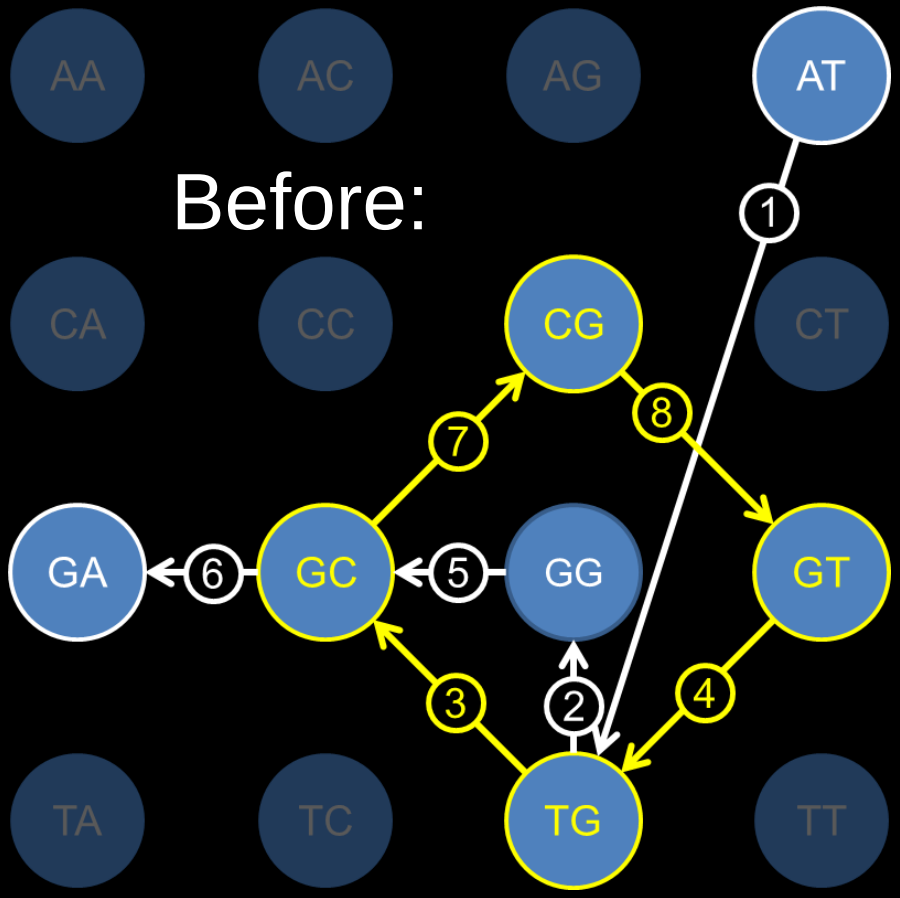
- Each node represents a prefix or suffix.
- Start with an edge with outdegree > indegree
- Assemble the contigs in edge order:

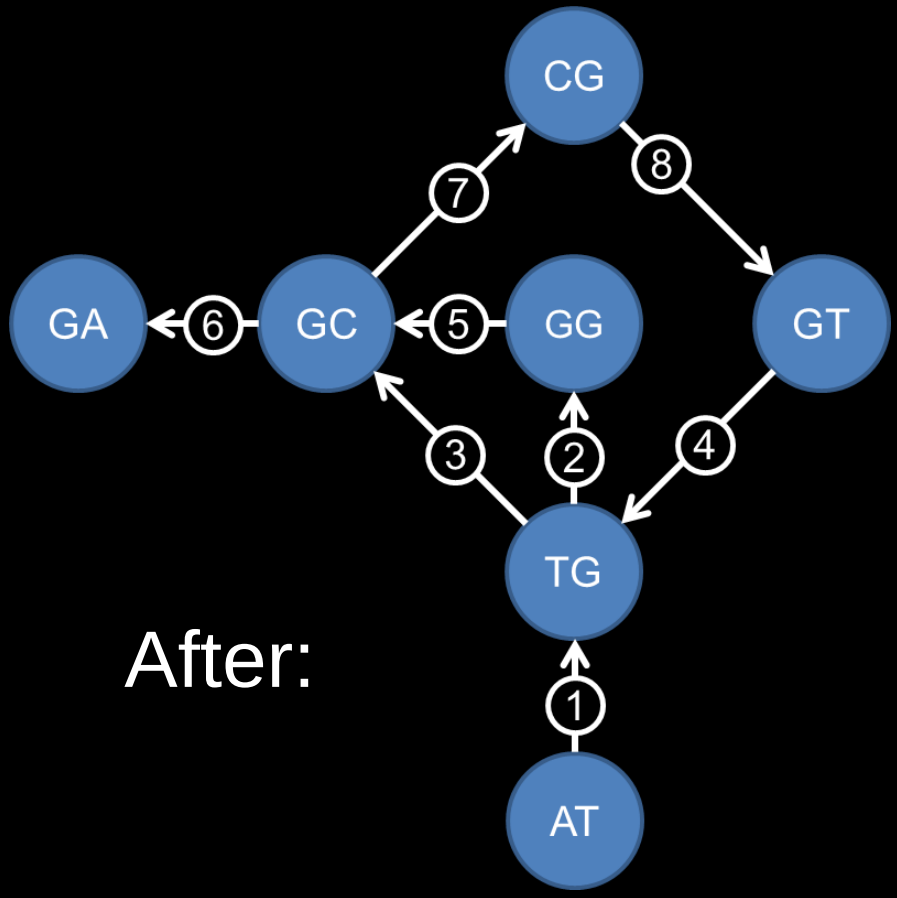
  ① ③ ⑥
- Here's a cycle

  ⑦ ⑧ ④ ③

A slight re-arrangement:

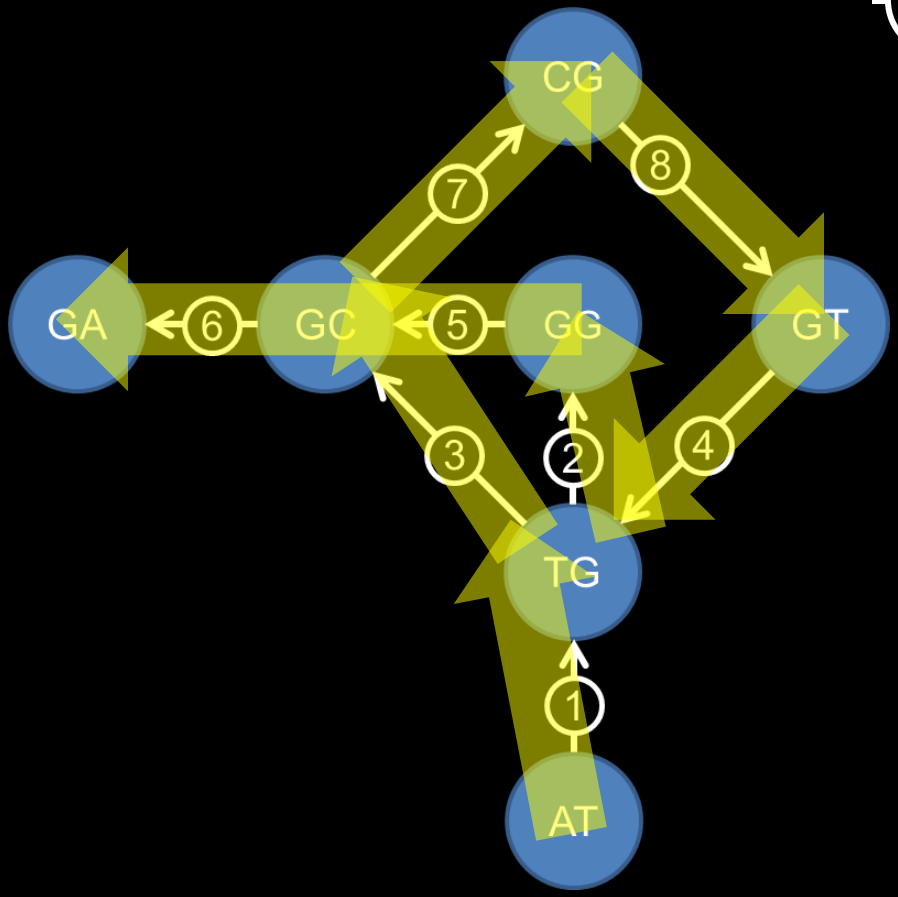
- I am just drawing the graph a different way, for the next slide. This change does not affect the graph at all.





Incorporating cycles into the sequence

- Cycles create ambiguity. You cannot know what order the cycles are in.
- One ordering:



Node Order:

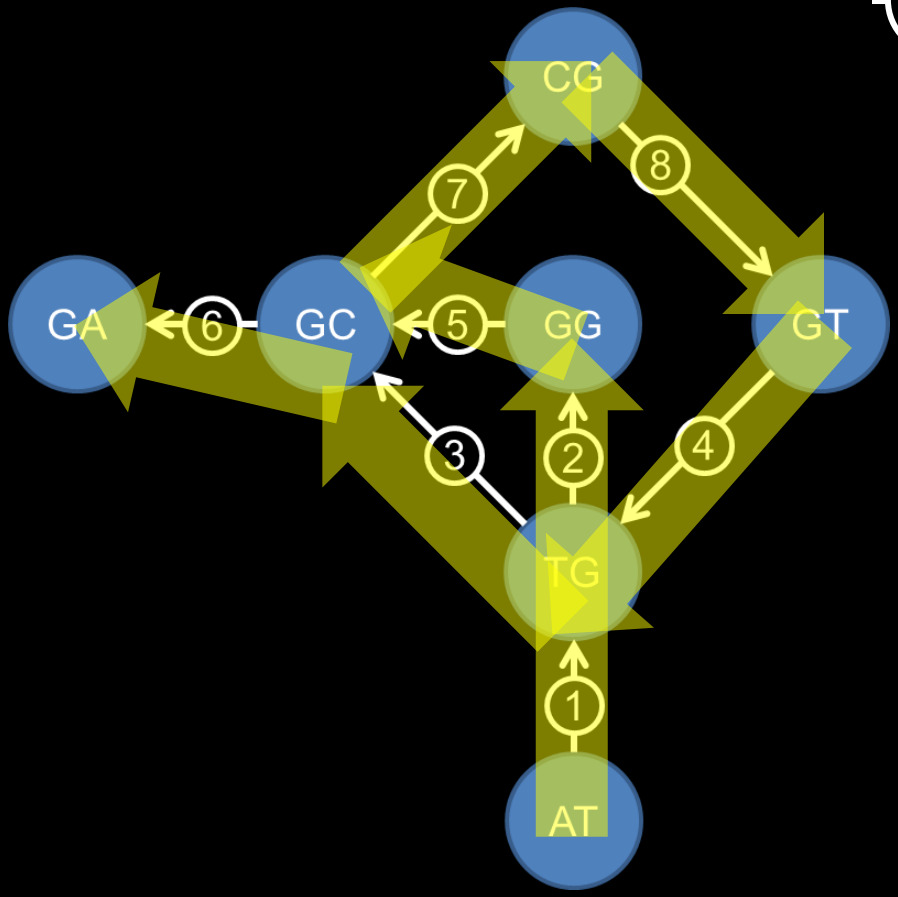
Edge Order:

Sequence Order:

```
1: ATG
3:  TGC
7:   GCG
8:    CGT
4:     GTG
2:      TGG
5:       GGC
6:        GCA
Final: ATGCGTGGCA
```

Incorporating cycles into the sequence

- Cycles create ambiguity. You cannot know what order the cycles are in.
- Second ordering:



Node Order:

Edge Order:

Sequence Order:

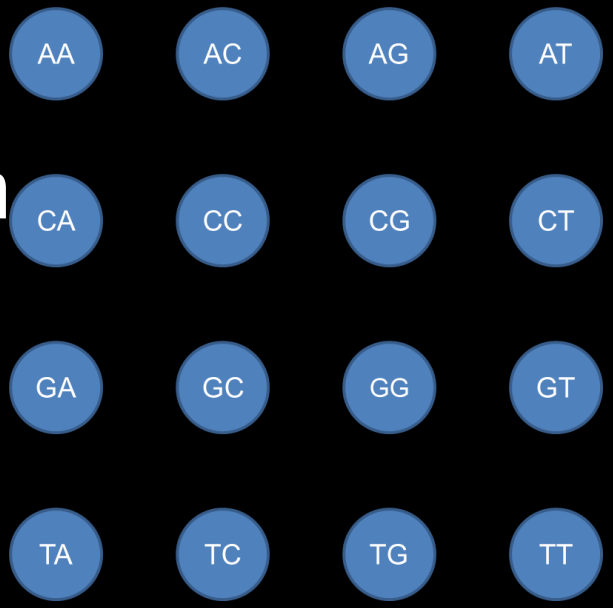
```
1: ATG
2:  TGG
5:   GGC
7:    GCG
8:     CGT
4:      GTG
3:       TGC
6:        GCA
Final: ATGGCGTGCA
```

The final output

- Two possible sequences:
  - **ATGCGTGGCA**
  - **ATGGCGTGCA**

- This is a very simple example.
- Project 1 differs in the following ways:
  1) Read overlaps will be 15 and not 2
  2) There will be many identical reads
  3) There will be more than 1x coverage

15 nucleotide read overlaps



- If there are only 2 nucleotides in any prefix or suffix, then there are only 4^2=16 possible nodes
- If there are 15 nucleotides in any prefix or suffix, then there are 4^{15}= 1,073,741,824 nodes
- If you store one integer for every node, that's 1 Gigabyte of memory
- SunLab machines only have 4GB of RAM

- This sounds like a serious problem..

Encoding a prefix into an integer

- There are four possibilities for each 15 character prefix/suffix
- Lets break the prefix/suffix down into characters 15 characters numbered n_0 - n_{14}
  - $n_0\ n_1\ n_2\ n_3\ n_4\ n_5\ n_6\ n_7\ n_8\ n_9\ n_{10}\ n_{11}\ n_{12}\ n_{13}\ n_{14}$
- Based on the characters ACGT, we assign n_i the value 0, 1, 2, 3, respectively:
  - A = 0, C = 1, G = 2, T = 3
- We can thus assign the entire prefix the following index V:

$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$

This value ranges from 0 to 4^{15}-1

Virtual representations

- Given the index V of a specific node, you'll need to know which edges go in and which go out of the node

- You have 60,000 reads, but you have one billion possible prefixes and suffixes.
  - The vast majority of prefixes and suffixes are not used. You don't need to store them all.

- You resolve this with Hashing, which we will discuss next time.

Identical Reads and multiple coverage

- For your projects you will not be dealing with a graph with just one edge between nodes, but often several.
- Building contigs out of these graphs will require that you have a labeling system that allows you to easily remember if you have used a particular read before
- Fortunately, Hashing allows you to deal with this issue easily as well

How to start a contig

- When you build your contig, you need to start at a node with outdegree greater than indegree.
- This ensures that you are starting at the beginning of a contig.
- If there are no nodes with outdegree greater than indegree, but there still are unused reads, then start at a node with indegree equal to outdegree
  - There should no longer exist any nodes with indegree > outdegree, either.

How to finish

- You will end up with many contigs that are similar or nearly identical to each other
  - Slight differences will occur at the ends of the contigs because the reads start at different places
- Eliminate duplicate contigs by looking for contigs that are nearly identical.

This is a big picture view of Assembly

- Now you understand how the implementation project will work, overall
- On Thursday I will tie up the loose ends of explaining how assembly works with 15 base prefixes and Hashing

Questions