# Bioinformática - TP 2 - Montagem de Genomas

**Alunos:**

- Francisco Galuppo Azevedo - 2017014960
- Mathias Coelho Batista - 2020722784
- Pedro Souza - 2013007684
- Ramon Durães - 2019720188
- Thaís Soares Lamas - 2017014910

## 1. Overlap-Layout-Consensus

Segue abaixo o código para a montagem de genomas utilizando a estratégia Overlap-Layout-Consensus (chamada de OLC abaixo).

In [1]:
```
from itertools import permutations
```

In [2]:
```
reads = ['TGGCA', 'GCATTGCAA', 'TGCAAT', 'CAATT', 'ATTTGAC']
dna_seq = 'TGGCATTGCAATTTGAC'
```

In [3]:
```
def overlap(a, b, min_len=3):
    inicio = 0
    while True:
        inicio = a.find(b[:min_len], inicio)
        if inicio == -1:
            return 0
        elif b.startswith(a[inicio:]):
            return len(a) - inicio
        inicio += 1

print(overlap('TGGCA', 'GCATTGCAA', 3))
```

```
3
```

In [4]:
```
def naive_overlap(reads, min_len):
    overlaps = dict()

    for a, b in permutations(reads, 2):
        overlap_len = overlap(a, b, min_len)

        if overlap_len > 0:
            overlaps[(a, b)] = overlap_len
    return overlaps

naive_overlap(reads, min_len=3)
```

Out[4]:
```
{('TGGCA', 'GCATTGCAA'): 3,
 ('GCATTGCAA', 'TGCAAT'): 5,
 ('GCATTGCAA', 'CAATT'): 3,
 ('TGCAAT', 'CAATT'): 4,
 ('CAATT', 'ATTTGAC'): 3}
```

In [5]:
```
```

```
def pegar_maximo_overlap(reads, min_len):
    readA, readB = None, None
    melhor_overlap = 0

    for a,b in permutations(reads, 2):
        overlap_len = overlap(a, b, min_len)

        if overlap_len > melhor_overlap:
            readA, readB = a, b
            melhor_overlap = overlap_len

    return readA, readB, melhor_overlap

pegar_maximo_overlap(reads, min_len=3)
```

Out[5]:
```
('GCATTGCAA', 'TGCAAT', 5)
```

In [6]:
```
def olc_guloso(reads, min_len):
    readA, readB, overlap_len = pegar_maximo_overlap(reads, min_len)

    while overlap_len > 0:
        reads.remove(readA)
        reads.remove(readB)
        reads.append(readA + readB[overlap_len:])

        readA, readB,overlap_len = pegar_maximo_overlap(reads, min_len)

    return ''.join(reads)

olc_guloso(reads, min_len=3)
```

Out[6]:
```
'TGGCATTGCAATTTGAC'
```

## 2. Grafos De Bruijn

A ideia principal do algoritmo é trabalhar com a sobreposição de k-mers. Para isso, as funções implementadas Vamos trabalhar somente com as funções `de_bruijn_graph` e `visualizar_de_bruijn` .

In [7]:
```
def de_bruijn_graph(dna_seq, kmer):
    vertices = set()
    arestas = list()

    for i in range(len(dna_seq) - kmer + 1):
        vertices.add(dna_seq[i: i+kmer-1])
        vertices.add(dna_seq[i+1: i+kmer])

        arestas.append((dna_seq[i:i+kmer-1], dna_seq[i+1: i+kmer]))

    return vertices, arestas
```

In [8]:
```
vertices, arestas = de_bruijn_graph(dna_seq, 4)
```

In [9]:
```
print(f'Os vértices são {vertices}')
```

```
Os vértices são {'GAC', 'GCA', 'TGC', 'TTG', 'TGA', 'CAA', 'ATT', 'CAT', 'GG
C', 'AAT', 'TGG', 'TTT'}
```

In [10]:
```
print(f'As arestas são {arestas}')
```

```
As arestas são [('TGG', 'GGC'), ('GGC', 'GCA'), ('GCA', 'CAT'), ('CAT', 'AT
T'), ('ATT', 'TTG'), ('TTG', 'TGC'), ('TGC', 'GCA'), ('GCA', 'CAA'), ('CAA',
'AAT'), ('AAT', 'ATT'), ('ATT', 'TTT'), ('TTT', 'TTG'), ('TTG', 'TGA'), ('TG
A', 'GAC')]
```

In [11]:
```
def visualizar_de_bruijn(dna_seq, kmer):
    vertices, arestas = de_bruijn_graph(dna_seq, kmer)

    dot_str = "digraph \"Debruijn graph\" {rankdir=\"LR\";\n"

    for v in vertices:
        dot_str += f' {v} [label="{v}"];\n'

    for fonte, destino in arestas:
        dot_str += f' {fonte} -> {destino};\n'

    dot_str += '}\n'
    return dot_str
```

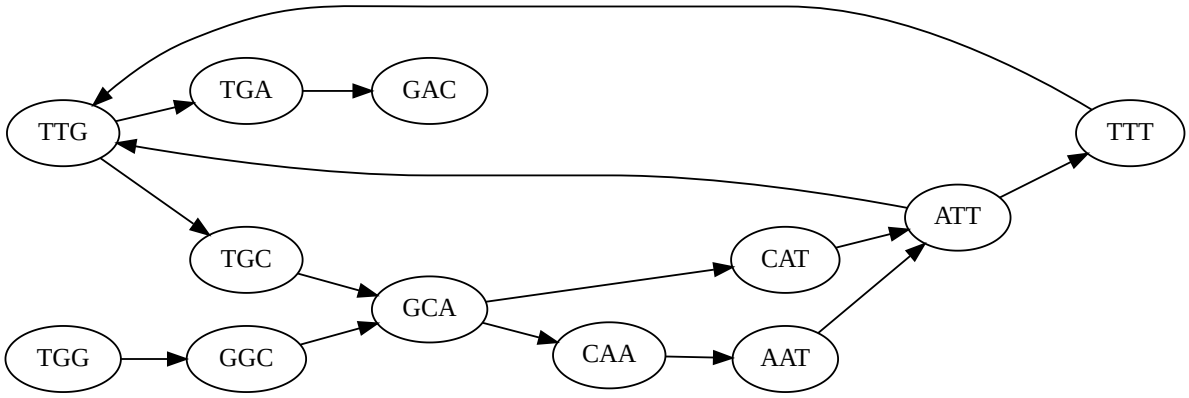
In [12]:
```
print(visualizar_de_bruijn(dna_seq, 4))
```

```
digraph "Debruijn graph" {rankdir="LR";
 GAC [label="GAC"];
 GCA [label="GCA"];
 TGC [label="TGC"];
 TTG [label="TTG"];
 TGA [label="TGA"];
 CAA [label="CAA"];
 ATT [label="ATT"];
 CAT [label="CAT"];
 GGC [label="GGC"];
 AAT [label="AAT"];
 TGG [label="TGG"];
 TTT [label="TTT"];
 TGG -> GGC;
 GGC -> GCA;
 GCA -> CAT;
 CAT -> ATT;
 ATT -> TTG;
 TTG -> TGC;
 TGC -> GCA;
 GCA -> CAA;
 CAA -> AAT;
 AAT -> ATT;
 ATT -> TTT;
 TTT -> TTG;
 TTG -> TGA;
 TGA -> GAC;
}
```

Para visualizar, vamos usar o comando dot de linux/unix

In [13]:
```
with open("dna_seq_4.txt", "w") as f:
    grafo = visualizar_de_bruijn(dna_seq, 4)
    f.write(grafo)
```

In [14]:
```
!dot -Tsvg dna_seq_4.txt > dna_seq_4.svg
```




**NOTA: Caso a imagem acima não apareça, favor visualizar a versão em PDF.**

# Extra: Melhoria de Código

## Função olc_guloso

Na função `olc_guloso` fornecida, as reads, passadas como parâmetro, são alteradas. Isso acontece devido à atribuição por referência que estava sendo feita anteriormente. Para melhorar isso, basta criar a variável `reads_copy = reads.copy()` . Desta forma, após rodar a função, a variável reads continua com seu valor original.

Isso foi implementado na função abaixo `olc_guloso_improved` .

In [15]:
```
# Demonstração da alteração da variável original
reads = ['TGGCA', 'GCATTGCAA', 'TGCAAT', 'CAATT', 'ATTTGAC']
print(olc_guloso(reads, min_len=3))
print(reads)
```

```
TGGCATTGCAATTTGAC
['TGGCATTGCAATTTGAC']
```

In [16]:
```
# Código melhorado
reads = ['TGGCA', 'GCATTGCAA', 'TGCAAT', 'CAATT', 'ATTTGAC']

def olc_guloso_improved(reads, min_len):
    readA, readB, overlap_len = pegar_maximo_overlap(reads, min_len)
    reads_copy = reads.copy()

    while overlap_len > 0:
        reads_copy.remove(readA)
        reads_copy.remove(readB)
        reads_copy.append(readA + readB[overlap_len:])

        readA, readB,overlap_len = pegar_maximo_overlap(reads_copy, min_len)

    return ''.join(reads_copy)

print(olc_guloso_improved(reads, min_len=3))
print(reads)
```

```
TGGCATTGCAATTTGAC
['TGGCA', 'GCATTGCAA', 'TGCAAT', 'CAATT', 'ATTTGAC']
```

## Funções de overlap

No código apresentado em aula, haviam duas funções que calculavam overlaps: uma para

todos e outra para o máximo overlap. Uma forma de melhorar estas funções é guardando o tamanho do máximo overlap na função que busca por todos.

Isso foi implementado abaixo, com a flag `only_max` indicando se deve-se retornar todos os overlaps encontrados ou apenas o maior. Dessa forma, as funções `naive_overlap` e `pegar_maximo_overlap` foram combinadas em uma mais eficiente que chamamos de `get_overlaps` .

In [17]:
```
def get_overlaps(reads, min_len, only_max=False):
    overlaps = dict()
    max_overlap_len = 0
    best = None

    for a, b in permutations(reads, 2):
        overlap_len = overlap(a, b, min_len)

        if overlap_len > 0:
            overlaps[(a, b)] = overlap_len
            if overlap_len > max_overlap_len:
                max_overlap_len = overlap_len
                best = (a, b, max_overlap_len)

    if only_max:
        return best
    else:
        return overlaps
```

In [18]:
```
get_overlaps(reads, min_len=3, only_max=False)
```

Out[18]:
```
{('TGGCA', 'GCATTGCAA'): 3,
 ('GCATTGCAA', 'TGCAAT'): 5,
 ('GCATTGCAA', 'CAATT'): 3,
 ('TGCAAT', 'CAATT'): 4,
 ('CAATT', 'ATTTGAC'): 3}
```

In [19]:
```
get_overlaps(reads, min_len=3, only_max=True)
```

Out[19]:
```
('GCATTGCAA', 'TGCAAT', 5)
```