```
def eulerian_cycle(self):
    if not self.check_balanced_graph(): return None
    edges_visit = list(self.get_edges())
    res = [ ]
    while edges_visit:
        pair = edges_visit[0]
        i = 1
        if res != []:
            while pair[0] not in res:
                pair = edges_visit[i]
                i = i + 1
        edges_visit.remove(pair)
        start, nxt = pair
        cycle = [start, nxt]
        while nxt != start:
            for suc in self.graph[nxt]:
                if (nxt, suc) in edges_visit:
                    pair = (nxt,suc)
                    nxt = suc
                    cycle.append(nxt)
                    edges_visit.remove(pair)
        if not res: res = cycle
        else:
            pos = res.index(cycle[0])
            for i in range(len(cycle)-1): res.insert(pos + i +1, cycle[i+1])
    return res
```

Implementando ciclos Eulerianos

(classe MyGraph)

Função para retornar ciclo Euleriano (se existir) usando algoritmo anterior

```
gr = MyGraph( {1:[2], 2:[3,1], 3:[4], 4:[2,5],
5:[6], 6:[4]} )
gr.printGraph()
print (gr.check_balanced_graph() )
print (gr.eulerian_cycle() )
```

Implementando caminhos Eulerianos
(classe MyGraph)

```
def check_nearly_balanced_graph (self):
        res = None, None
        for n in self.graph.keys():
                indeg= self.in_degree(n)
                outdeg= self.out_degree(n)
                if indeg - outdeg == 1 and res[1] is None: res = res[0], n
                elif indeg - outdeg == -1 and res[0] is None: res = n, res[1]
                elif indeg == outdeg: pass
                else: return None, None
        return res
```

Função para verificar se o grafo é semi-balanceado

Implementando caminhos Eulerianos
(classe MyGraph)

Função para retornar caminho *Euleriano* (se existir) usando algoritmo anterior

```
def eulerian_path (self):
        unb = self.check_nearly_balanced_graph()
        if unb[0] is None or unb[1] is None: return None
        self.graph[unb[1]].append(unb[0])
        cycle = self.eulerian_cycle()
        for i in range(len(cycle)-1):
                if cycle[i] == unb[1] and cycle[i+1] == unb[0]:
                        break
        path = cycle[i+1:] + cycle[1:i+1]
        return path
```

```
gr = MyGraph( {1:[2], 2:[3,1], 3:[4], 4:[2,5],  5:[6], 6:[]} )
gr.printGraph()
print (gr.check_balanced_graph() )
print (gr.check_nearly_balanced_graph()  )
print (gr.eulerian_path() )
```

Implementando caminhos Eulerianos
(classe DeBruijnGraph)

Necessário alterar in degree para grafos com arcos repetidos

```
def in_degree(self, v):
    res = 0
    for k in self.graph.keys():
        if v in self.graph[k]:
            res += self.graph[k].count(v)
    return res
```

```
def test2():
    frags = [ "ACC", "ATA", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA",
"GCA", "GGC", "TAA", "TCA", "TGG", "TTC", "TTT"]

    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()
    print (dbgr.check_nearly_balanced_graph())
    print (dbgr.eulerian_path())

test2()
```

Implementando caminhos Eulerianos: fechando o ciclo

```
def test3():
        orig_sequence = "ATGCAATGGTCTG"
        frags = composition(3, orig_sequence)
        …

test3()
```

```
def seq_from_path (self, path):
        seq= path[0]
        for i in range(1,len(path)):
                nxt = path[i]
                seq += nxt[-1]
        return seq
```

Vamos assumir que sabemos a sequência original e verificar se conseguimos recuperá-la !

Complete o código !!

Implementando caminhos Eulerianos: fechando o ciclo

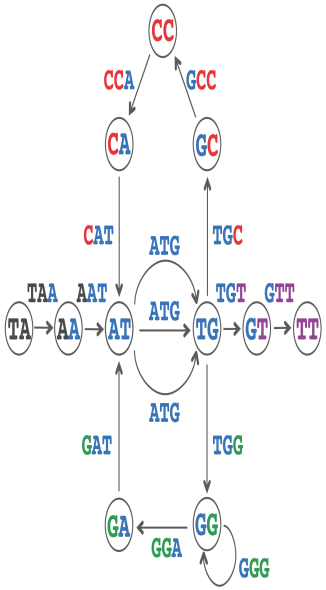
```
def test3():
        orig_sequence = "ATGCAATGGTCTG"
        frags = composition(3, orig_sequence)
        dbgr = DeBruijnGraph(frags)
        dbgr.print_graph()
        print (dbgr.check_nearly_balanced_graph())
        p= dbgr.eulerian_path()
        print (p)
        print (dbgr.seq_from_path(p))

test3()
```

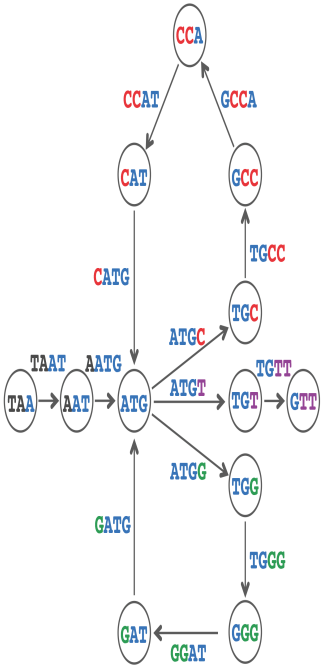
```
def seq_from_path (self, path):
        seq= path[0]
        for i in range(1,len(path)):
                nxt = path[i]
                seq += nxt[-1]
        return seq
```

Teste com outras alternativas … será que conseguimos sempre identificar corretamente a sequência original ?
O que é que isto implica ?
Verifique o que acontece quando aumenta o tamanho da sequência !!

Grafos de DeBruijn no mundo real da sequenciação

Em casos reais, quanto maior for o tamanho das leituras do sequenciador menor é o grau médio dos nós e mais fácil é definir o caminho Euleriano (ver exemplo de 3 grafos para a mesma sequência original com diferentes tamanhos de fragmentos)



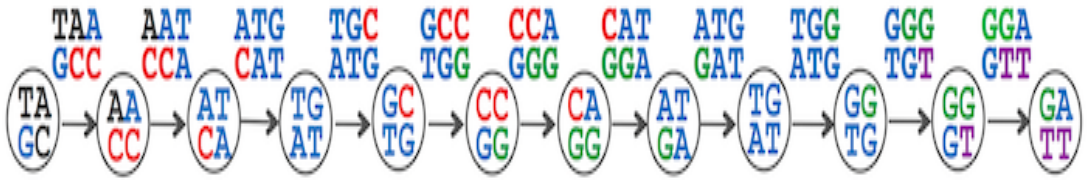


Grafos de DeBruijn no mundo real da sequenciação

Repetições no genoma causam problemas, especialmente se forem maiores que o tamanho das leituras

Uma forma de ultrapassar este problema são as **leituras emparelhadas (*paired reads*)**, onde temos k nucléotidos medidos, um gap fixo de d nuceótidos e outros k nucleótidos medidos

Algoritmos semelhantes aos vistos atrás com grafos de DeBruijn podem ser usados para estas leituras emparelhadas

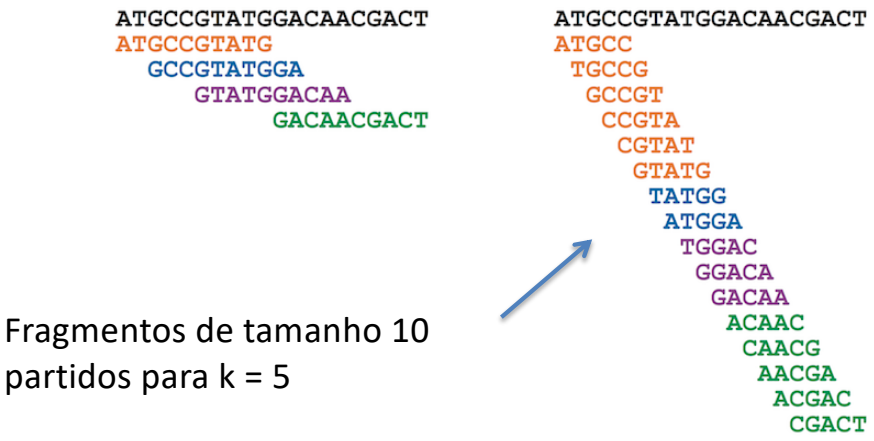


Problemas no mundo real da sequenciação

Numa sequenciação, cada fragmento do genoma é sequenciado um dado nº de vezes, chamado de **cobertura**

No entanto, este é um valor médio havendo fragmentos que não aparecem

Uma forma de lidar com este problema é trabalhar nos algoritmos com valores de k mais pequenos que o tamanho dos fragmentos e partir as leituras originais



Fragmentos de tamanho 10 partidos para k = 5

Problemas no mundo real da sequenciação: contigs

Mesmo com a solução anterior, continuarão a existir gaps, levando os grafos de DeBruijn a terem arcos em falta

Assim, programas de montagem de genomas, numa primeira fase, montam partes contíguas do genoma (designadas por **contigs**) com o menor número e a maior extensão possível

Numa sequenciação procariótica, é típico termos centenas de contigs como resultado, com tamanhos na ordem dos milhares até às centenas de milhares de nucleótidos

Problemas no mundo real da sequenciação: erros de leitura

A existência de **erros** de leitura em muitos dos fragmentos lidos torna mais difícil a tarefa da montagem

Alguns erros mais simples (e.g. troca de um nucleótido) podem ser identificados nos grafos de DeBruijn e removidos por algoritmos especializados

No entanto, é sempre complexo saber quais os caminhos no grafo que são erros e os que são corretos

A existência de repetições não exatas no genoma torna este problema ainda mais complexo