Grafos e sequenciação de genomas

Como os grafos e os algoritmos sobre grafos podem ajudar na montagem de leituras (fragmentos) de genomas

Parcialmente adaptado do curso “Bioinformatics Algorithms”, P. Pevzner et al
How Do We Assemble Genomes? (Graph Algorithms)

Implementação composição em k-mers

```
def composition(k, seq):
    creat list res
    for each character positon in seq until size of seq – k + 1:
        append string of char position until char position + k
    sort the list res
    return res
```

Retorna uma lista de strings consecutivas com tamanho k a partir da sequência

```
def test1():
        seq = "CAATCATGATG"
        k = 3
        print (composition(k, seq))

test1()
Result:
['CAA', 'AAT', 'ATC', 'TCA', 'CAT', 'ATG', 'TGA', 'GAT', 'ATG']
```

Implementando grafo de DeBruijn

```
from MyGraph import MyGraph

class DeBruijnGraph (MyGraph):

    def __init__(self, fragments):
        MyGraph.__init__(self, {})
        self.createDeBruijnGraph(fragments)

    def add_edge(self, o, d):
        if o is not in graph keys:
            add node o;
        if d is not in graph keys:
            add node d
        append d to successors of o

        def  createDeBruijnGraph(self, kmers):
            ...
```

Como anteriormente, vamos criar uma classe **DeBruijnGraph** para representar estes grafos que é sub-classe de ***MyGraph***

Método ***add_edge*** redefinido para admitir arcos repetidos, i.e. múltiplos arcos entre o mesmo par de nós

Implementando grafo de DeBruijn

```
def createDeBruijnGraph(self, kmers):
    for each sequence in kmers:
        get suffix of sequence
        add suffix as node
        get prefix of sequence
        add prefix as node
        create a edge prefix, suffix
```

```
def test2():
    frags = ["AAT", "ATG", "GTT", "TAA", "TGT"]
    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()

def test3():
    frags =["AAT", "ATG", "ATG", "ATG", "CAT", "CCA", "GAT", "GCC", "GGA", "GGG","GTT",
            "TAA", "TGC", "TGG", "TGT"]
    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()
```

Implementando ciclos Eulerianos
(classe MyGraph)

```
def checkBalancedNode(self, node):
        return degree in of node is equal to out degree node

def checkBalancedGraph(self):
        for each node in graph nodes:
                if node is not balanced: return false
        return True
```

Funções para verificar se um nó é balanceado e se o grafo é balanceado

Implementando ciclos Eulerianos (classe MyGraph)

```
def eulerianCycle(self):
        if graph is not balanced: return None
        create a list of all graph edges to visit
        create a empty list res
        while edges to visit list is not empty:
                get first edge as current edge from list of edges
                create index i equal to 1
                if res is not empty:
                        while first node of edge is not in res list:
                                set current edge equal to edge at i position
                                increase index i by 1
                remove current edge from to visit edges list
                define start node and next node from current edge
                create a list cycle populated with edge nodes
                while next node is not equal to start node:
                        for each successor of next node:
                                if next node and successor node edge is in edges to visit list:
                                        define current edge equal to next node and successor node edge
                                        define next node equal to successor node
                                        append next node to cycle list
                                        remove current edge from edges to visit list
                if res is empty:
                        define res equal to cycle list
                else:
                        get positionfirst in res of first element of cycle list
                        for index in range of 1 until size cycle:
                                insert cycle[i] to res at positionfirst + index
        return res
```

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

Implementando ciclos Eulerianos

```
def test4():
        gr = MyGraph( {1:[2], 2:[3,1], 3:[4], 4:[2,5], 5:[6], 6:[4]} )
        gr.print_graph()
        print(gr.checkBalancedGraph() )
        print(gr.eulerianCycle() )

Result:
1 -> [2]
2 -> [3, 1]
3 -> [4]
4 -> [2, 5]
5 -> [6]
6 -> [4]
True
[1, 2, 3, 4, 5, 6, 4, 2, 1]
```

Implementando caminhos Eulerianos
(classe MyGraph)

Função para verificar se o grafo é semi-balanceado

```
def checkNearlyBalancedGraph (self):
    create a tuple result that has saved_left_node and saved_right_node with None, None
    for each node in graph:
        get indegree of node
        get outdegree of node
        if indegree – outdegree is equal to 1 and saved_right_node is none:
            redefine result equal to saved_left_node, node
        else if indegree – outdegree is equal to -1 and saved_left_node is none:
            redefine result equal to node, saved_right_node
        else if indegree is equal to outdegree:
            continue the loop
        else:
            return None, None
    return result
```

Implementando caminhos Eulerianos
(classe MyGraph)

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

```
def eulerianPath (self):
        check if graph is neary balanced and define the saved_left_node and saved_right_node
        if saved_left_node is a none or saved_right_node is a none: return None
        create a edge where saved_left_node is successor of saved_right_node
        create a variable cycle with eurilan cycle method
        for each position in range of cycle size -1:
                if cycle on position is equal to saved_right_node and cycle position + 1 is equal to saved_left_node :
                        then break the loop;
        get the path using the resultant position from loop and the cycle list.
                    Note: join two lists:
                      the first list contains elements from cycle that are in loop position + 1 to end of cycle list.
                      the second list contains elements of cycle 1 to loop position + 1 .
        return path
```

Implementando caminhos Eulerianos

```
def test5():
        gr = MyGraph( {1:[2], 2:[3,1], 3:[4], 4:[2,5],  5:[6], 6:[]} )
        gr.print_graph()
        print(gr.checkBalancedGraph() )
        print(gr.checkNearlyBalancedGraph()  )
        print(gr.eulerianPath() )

Result:
1 -> [2]
2 -> [3, 1]
3 -> [4]
4 -> [2, 5]
5 -> [6]
6 -> []
False
(4, 6)
[4, 2, 1, 2, 3, 4, 5, 6]
```

Implementando caminhos Eulerianos
(classe DeBruijnGraph)

Necessário alterar in degree para grafos com arcos repetidos

```
def inDegree(self, v):
    define res equal to 0
    for each node in graph:
        if v is in successors of node:
            sum to res the count of v repeats as successor of node
    return res
```

```
def test6():
    frags = ["AAT", "ATG", "ATG", "ATG", "CAT", "CCA", "GAT", "GCC", "GGA", \
    "GGG", "GTT", "TAA", "TGC", "TGG", "TGT"]
    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()
    print (dbgr.checkNearlyBalancedGraph())
    print (dbgr.eulerianPath())
```

Implementando caminhos Eulerianos: fechando o ciclo

```
def test7():
        orig_sequence = "ATGCAATGGTCTG"
        frags = composition(3, orig_sequence)
        dbgr = DeBruijnGraph(frags)
        dbgr.print_graph()
        print (dbgr.checkNearlyBalancedGraph())
        p= dbgr.eulerianPath()
        print (p)
```

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

Teste com outras alternativas … será que conseguimos sempre identificar corretamente a sequência original ?
O que é que isto implica ?

Implementação grafo de sobreposições

Vamos implementar uma classe para representar grafos de sobreposições
Esta classe – ***OverlapGraph*** - será uma sub-classe da classe MyGraph para representar grafos orientados

```
class OverlapGraph(MyGraph):
        def __init__(self, fragments):
                MyGraph.__init__(self, {})
                self.createOverlapGraph(fragments)

        def createOverlapGraph(self, lseqs):
                ## add vertices
                ...

                ## add edges
                ...
```

```
def test8():
        frags = ["AAT", "ATG", "GTT", "TAA", "TGT”]
        ovgr = OverlapGraph(frags)
        ovgr.print_graph()

Result:
AAT -> ['ATG’]
ATG -> ['TGT']
GTT -> []
TAA -> ['AAT']
TGT -> ['GTT']
```

Implementando grafo de sobreposições

Adiciona as sequencias como nós no grafo, cria arcos caso haja alinhamento parcial de duas sequências

```
def createOverlapGraph(self, lseqs):
        ## add vertices
        for each seq in lseqs:
                add node seq to graph
        ## add edges
        for each seq in lseqs:
                get suffix of seq
                for each seq2 in lseqs:
                  if prefix of seq2 is equal to suffix:
                     add edge seq, seq2
```

Qual o problema desta solução?

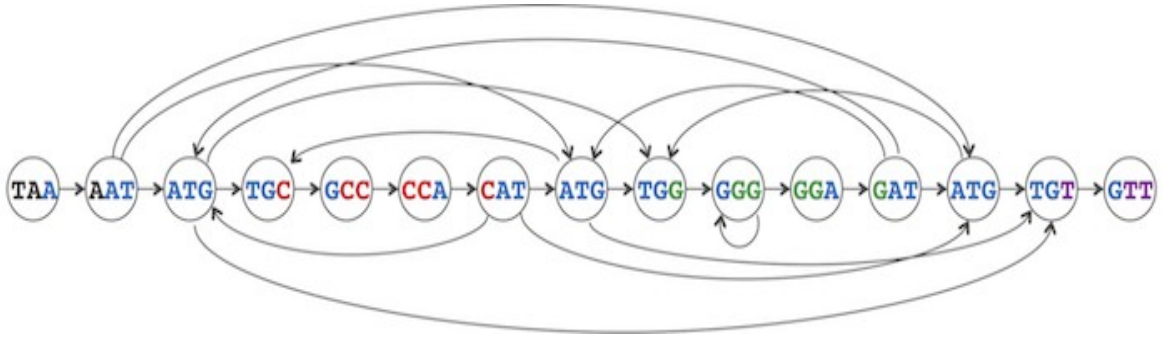
```
## auxiliary functions
def suffix (seq):
        return seq without first char

def prefix (seq):
        return seq without last char
```

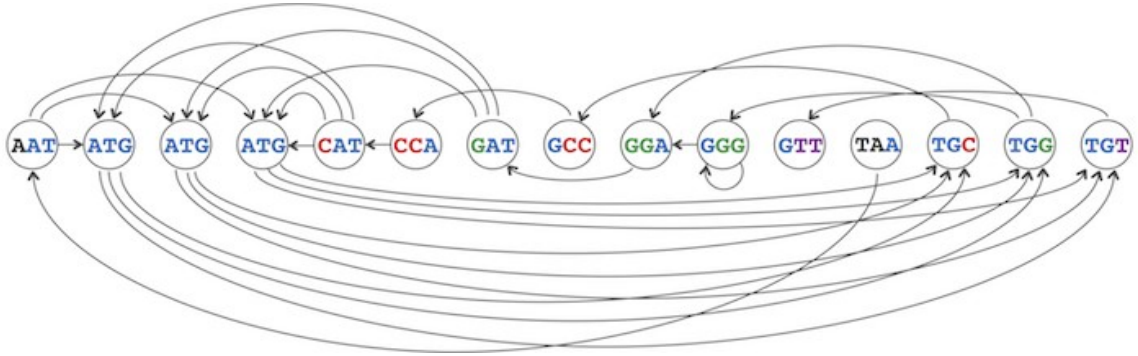
Funções auxiliares definidas fora da classe:
Suffix retorna a sequência sem o primeiro caracter;
Prefix retorna a sequência sem o ultimo caracter;

Implementando grafo de sobreposições - exemplo com repetições

```
def test9():
        frags = ["AAT","ATG","ATG","ATG","CAT","CCA","GAT","GCC","GGA", \
        "GGG","GTT","TAA","TGC","TGG","TGT"]
        ovgr = OverlapGraph(frags)
        ovgr.print_graph()
```



Seqs repetidas serão representadas por um único nó! Criará loops e muitos ciclos.



Implementando grafo de sobreposições com repetições

Adiciona as sequências como nós no grafo mas com numeração, cria arcos caso haja alinhamento parcial de duas sequências

```
def createOverlapGraphReps (self, lseqs):
        create id equal to 1
        for each seq in lseqs:
                add node with seq-id string
                increase id with 1
        reset id to 1
        for each seq in lseqs:
                get sequence suffix
                for each seq2 in lseqs:
                        if prefix of seq2 is equal to suffix:
                                for each instance in getinstances of seq2:
                                        add edge of seq-id string, instance
                        increase id with 1
```

```
def getInstances (self, seq):
    create list res
    for each node in  graph:
        if seq is in node:
            add seq to res
    return res
```

Como adicionamos numeração as sequências, temos que criar a função getInstances para encontrar a seq nos nós

Implementando grafo de sobreposições com repetições

```
def __init__(self, fragments, reps = True):
        MyGraph.__init__(self, {})
        if reps: call createOverlapGraphReps for fragmets
        else: call createOverlapGraph for fragments
        self.reps = reps
```

Modifica o init da classe para que seja permitida a escolha do método de criação de grafos de sobreposições

```
def test10():
        frags = ["AAT","ATG","ATG","ATG",...]
        ovgr = OverlapGraph(frags, True)
        ovgr.print_graph()
```

Implementando caminhos Hamiltonianos

Funções para verificar se caminho é correto e se caminho é Hamiltoniano:
Adicionadas a classe ***MyGraph***

```
def checkIfValidPath(self, p):
        if first p node not exist in graph: return False
        for second node on p until last node of p:
            if node is not on graph or node p is not successor of previous p node:
                return false
        return true
```

Retorna um booleano que garante se o caminho p é válido ou não no grafo

Implementando caminhos Hamiltonianos

Caminhos Hamiltonianos são **caminhos válidos** num grafo e que percorrem o **grafo completo** passando por todos os nós **uma única vez**!

```
def checkIfHamiltonianPath(self, p):
        if not p is not a valid path: return False
        create list of graph nodes to visit one single time
        if size of p is not equal to list to visit: return False
        for each node on p:
                if node is not on list of node to visit: return False
                remove node on list of nodes to visit

        if list of nodes to visit is not empty: return False
        return True
```

Retorna um booleano que garante se o caminho p é Hamiltoniano ou não no grafo

Função para verificar se caminho é Hamiltoniano:
Adicionada a classe ***MyGraph***

Implementando caminhos Hamiltonianos

```
def test11():
        frags = ["AAT", "ATG", "ATG", "ATG", "CAT", "CCA", "GAT", "GCC", "GGA", "GGG", \
        "GTT", "TAA", "TGC", "TGG", "TGT"]
        ovgr = OverlapGraph(frags, True)
        path = ["AAT-1", "ATG-4", "TGC-13"]
        print (ovgr.checkIfValidPath(path))
        print (ovgr.checkIfHamiltonianPath(path))
        path2 = ["TAA-12", "AAT-1", "ATG-2", "TGC-13", "GCC-8", "CCA-6", "CAT-5", "ATG-3", \
        "TGG-14", "GGG-10", "GGA-9", "GAT-7", "ATG-4", "TGT-15", "GTT-11"]
        print (ovgr.checkIfValidPath(path2))
        print (ovgr.checkIfHamiltonianPath(path2))
```

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def seqFromPath (self, p):
        If p is not Hamiltonian: return None
        create empty string res
        for each node on p:
                get set of p
                if res is empty:
                        add seq to res
                else:
                        add only last char of seq
        return seq
```

Retorna a sequência construída a partir de um caminho p

```
def getSeq (self, node):
        if node is not in graph nodes: return None
        if graph contains reps: return seq without numeration
        else: return seq from node
```

Função para dar a sequência reconstruída dado um caminho no grafo

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def test12():
        frags = ["AAT", "ATG", "ATG", "ATG", "CAT", "CCA", "GAT", "GCC", "GGA", "GGG","GTT",
                 "TAA", "TGC", "TGG", "TGT"]
        ovgr = OverlapGraph(frags, True)
        path2 = ["TAA-12", "AAT-1", "ATG-2", "TGC-13", "GCC-8", "CCA-6", "CAT-5", "ATG-3","TGG-14",
                "GGG-10", "GGA-9", "GAT-7", "ATG-4", "TGT-15", "GTT-11"]
        print (ovgr.checkIfHamiltonianPath(path2))
        print (ovgr.seqFromPath(path2))

Result:
AAT-1 -> []
ATG-2 -> []
...
TGC-28 -> ['GCC-8', 'GCC-23']
TGG-29 -> ['GGA-9', 'GGA-24', 'GGG-10', 'GGG-25']
TGT-30 -> ['GTT-11']
False
None
```

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def test13():
        frags = ["AAT", "ATG", "GTT", "TAA", "TGT"]
        ovgr = OverlapGraph(frags, False)
        ovgr.print_graph()
        print (ovgr.seqFromPath(["TAA","AAT","ATG","TGT","GTT"]))
        print (ovgr.seqFromPath(["TAA","TGT","ATG","TGT","GTT"]))

Result:
AAT -> ['ATG']
ATG -> ['TGT']
GTT -> []
TAA -> ['AAT']
TGT -> ['GTT']
TAATGTT
None
```

Implementando caminhos Hamiltonianos: procura exaustiva

Função para procura de caminhos *Hamiltonianos*: por nó
Adicionada na classe **MyGraph**

```
def searchHamiltonianPathFromNode(self, node):
    ...
```

Algoritmo implementa uma "árvore de procura" representada por 3 variáveis:

current – **nó** a processar
respath – **lista** que mantém caminho atual
visited – **dicionário** que mantém estado dos nós/arcos já explorados
(**chave**-nó; **valor**-índice do sucessor a explorar a seguir)

Backtracking é conseguido utilizando o respath como uma **stack** e o dicionário como **auxiliar de memória** para decidir o seguinte sucessor a procurar

Implementando caminhos Hamiltonianos: procura exaustiva

```
def searchHamiltonianPathFromNode(self, node):
    define current with node
    populate respath with node
    populate visited dict with node (key) and successor index (value) as 0
    while size of path is smaller than size of graph nodes:
        get current node successor index form visited dict
        if current node successor index is smaller than size node successors of current node:
            get the successor (using the successor index and current node from graph);
            increase the successor index on visited dictionary;
            if successor node is not in respath:
                append node to respath;
                add successor (key) to visited dictionary with index (value) as 0
                set current node equal to successor

        else:
            if size of respath is bigger than 1:
                remove last node of respath and get the removed node
                delete the removed node (key) from visited dictionary
                set current nod equal to last respath node
            else: return None
    return respath
```

Caso em que nó é adicionado ao caminho

Backtracking: recuar para testar caminhos alternativos

Implementando caminhos Hamiltonianos: procura exaustiva

```
def searchHamiltonianPath(self):
        for node in graph nodes:
                get Hamiltonian path from node
                if path is not none:
                        return path
        return None
```

Função para procura de caminhos *Hamiltonianos*: em todo o grafo
Adicionada na classe **MyGraph**

Implementando caminhos Hamiltonianos: procura exaustiva

```
def test14():
        frags = ["AAT", "ATG", "ATG", "ATG", "CAT", "CCA", "GAT", "GCC", "GGA", "GGG",
                        "GTT", "TAA", "TGC", "TGG", "TGT"]
                ovgr = OverlapGraph(frags, True)
        print (ovgr.searchHamiltonianPathFromNode("AAT-1"))
        print (ovgr.searchHamiltonianPathFromNode("TAA-12"))
        path = ovgr.searchHamiltonianPath()
        print (ovgr.checkIfHamiltonianPath(path))
        print (ovgr.seqFromPath(path))
```

Implementando caminhos Hamiltonianos: fechando o ciclo

```
def test15():
        orig_sequence = "CAATCATGATGATGATC"
        frags = composition(3, orig_sequence)
        ovgr = OverlapGraph(frags, True)
        path = ovgr.searchHamiltonianPath()
        print (path)
        print (ovgr.seqFromPath(path))
```

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

Teste com outras alternativas … será que conseguimos sempre identificar corretamente a sequência original ?
O que é que isto implica ?

Aumente o tamanho da sequência sem mudar o k … o que verifica ?
O que acontece se o k aumentar ?