Grafos e sequenciação de genomas

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

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

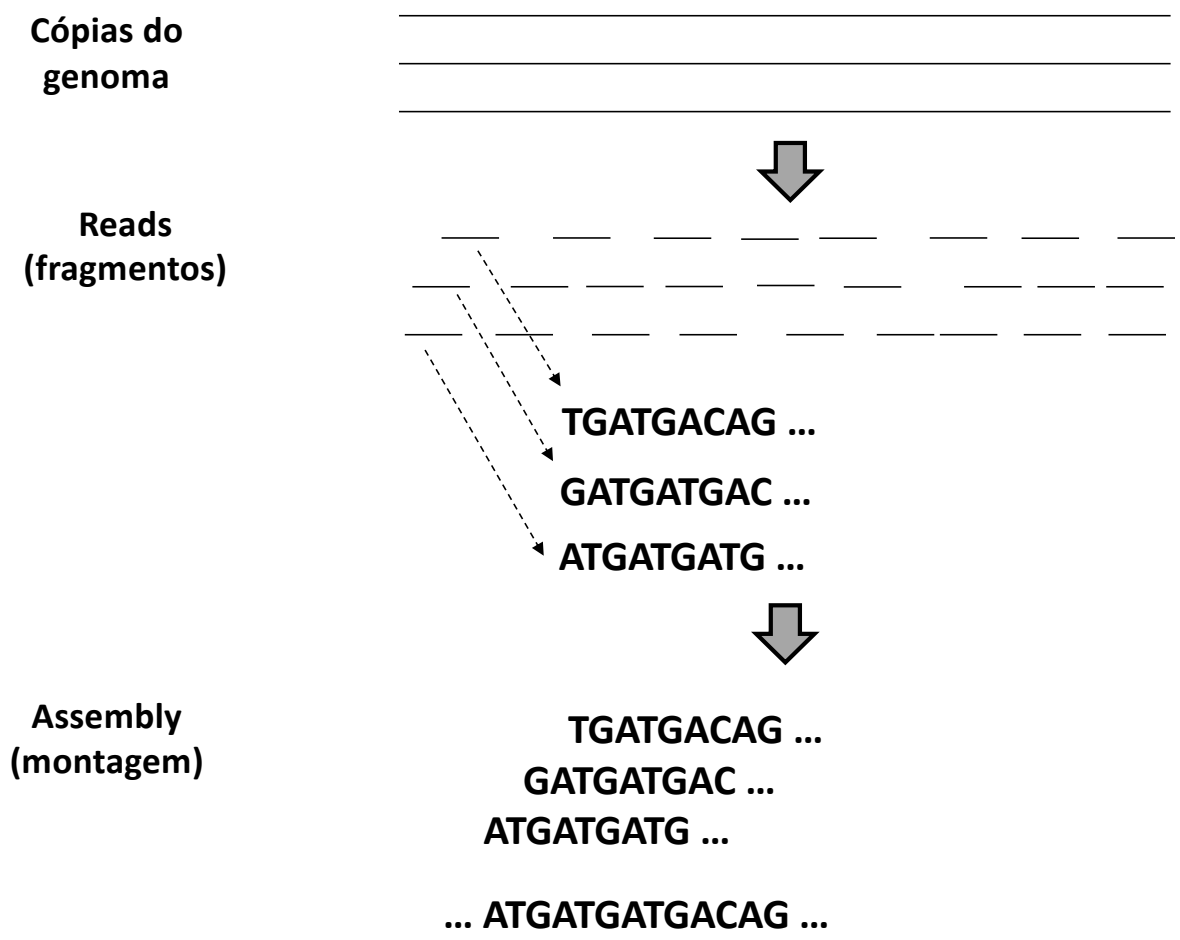
Sequenciação de genomas

Sequenciar um genoma não é como ler um livro pois não existem tecnologias que permitam sequenciar um genoma do início até ao final

As técnicas de sequenciação criam conjuntos de leituras (*reads*) de pequenos **fragmentos** do DNA com um máximo de algumas centenas de bases

É necessário haver algoritmos e ferramentas computacionais para **montar estes fragmentos** reconstituindo o genoma original que tem tipicamente centenas ou milhares de milhões de bases

Sequenciação de genomas



Montagem do genoma

Localização dos segmentos não é conhecida pelo que a montagem tem que ser realizada pela **sobreposição** dos fragmentos (tal como na montagem de um puzzle)

A **montagem** (*assembly*) do genoma a partir dos fragmentos constitui um problema computacional de **elevada complexidade**, dado a pequena dimensão dos fragmentos quando comparada com a grande dimensão dos genomas completos

Até aos anos 1990, pensava-se ser impossível montar genomas completos de organismos mais complexos; mesmo hoje, os genomas de maior dimensão estão ainda fora do alcance dos programas mais modernos de montagem de genomas

Montagem do genoma

Outros fatores tornam o problema real ainda mais difícil:

- As moléculas de DNA têm duas **cadeias complementares** e não há forma de saber de qual delas é proveniente cada fragmento
- Os sequenciadores têm uma dada taxa de **erro**; estes tornam o processo de sobreposição mais difícil pois em muitos casos não é perfeito
- Há regiões do genoma que são difíceis ou impossíveis de sequenciar, não sendo cobertas por nenhum fragmento

Numa primeira fase dos nossos algoritmos, assumiremos que estes problemas não existem para desenvolver algoritmos para **casos ideais** de cobertura total do genoma e inexistência de erros, assumindo que todas as leituras são feitas numa única cadeia

Composição de uma string em *k-mers*

Uma definição útil nos algoritmos que se seguem é a da **composição** de uma string (sequência) *s* em segmentos de tamanho *k* (**k-mers**) – ***comp$_k$(s)***

Esta é definida como a coleção de todas as sub-sequências de *s*, de tamanho *k*, incluindo repetições

Dado que, na sequenciação, não temos ideia da ordem dos fragmentos, vamos ordenar os segmentos por ordem lexicográfica, i.e. pela ordem "alfabética" que apareceriam num dicionário

Implementação composição em k-mers

```
def composition(k, seq):
    ...
```

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

test1()
```

Resultado: CAA, AAT, ATC, TCA, CAT, ATG, TGA, GAT, ATG

Implementação composição em k-mers

```
def composition(k, seq):
    res = [ ]
    for i in range(len(seq)-k+1):
        res.append(seq[i:i+k])
    res.sort()
    return res
```

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

test1()
```

Problema da reconstrução de uma string a partir da sua composição

Em sequenciação, não sabemos a sequência original, mas conhecemos a sua composição em fragmentos

Assim, um problema mais útil é o problema inverso: **reconstruir a string original dada a sua composição em k-mers**

- Entradas do problema
  - O valor de *k*; uma coleção *p* de k-mers
- Saída do problema
  - Uma string *s* com composição de k-mers igual a *p* (se existir)

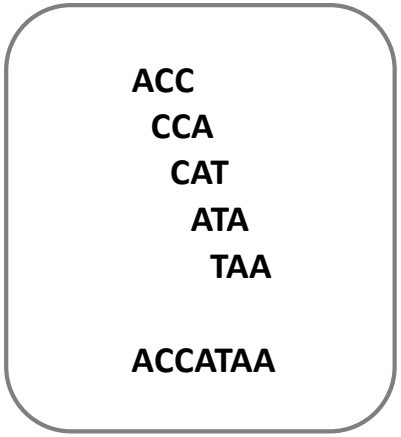
Problema da reconstrução de uma string a partir da sua composição

A solução mais natural para este problema será considerar um dos fragmentos para começar e estendê-lo (numa das duas direções) com fragmentos que tenham sobreposição de k-1 posições

- Vamos testar esta estratégia com o exemplo seguinte:
  ACC ATA CAT CCA TAA
- Teste agora com o exemplo:
  ACC AGT ATA ATT CAT CCA GTA TAA TAG TAT TTA

Será este um algoritmo viável ?

Problema da reconstrução de uma string da sua composição



Correto: usa todos os fragmentos

Problema da reconstrução de uma string da sua composição

Chegar à solução correta pode implicar “back-tracking”

As repetições tornam o problema da montagem do genoma mais complexo

Uma ilustração do problema das repetições na montagem do genoma pode ser encontrada no puzzle “triazzle”, onde padrões repetidos em várias zonas do puzzle tornam complexa a tarefa de definir o local correto de cada peça !

Problema da reconstrução de uma string: definições

Definições necessárias:

- **Prefixo$_{k-1}$** de uma sequência de tamanho k: primeiros k-1 elementos da sequência
- **Sufixo$_{k-1}$** de uma sequência de tamanho k: últimos k-1 elementos da sequência

Dadas estas definições podemos ligar duas sequências S1 e S2 como consecutivas se o **sufixo$_{k-1}$ de S1 é igual ao prefixo$_{k-1}$ de S2**

- Exemplo: TAA -> AAT dado que prefixo(AAT) = sufixo (TAA) = AA

Problema da reconstrução de uma string: grafo de sobreposições

Dado um conjunto de sequências (ou fragmentos) podemos criar o respetivo **grafo de sobreposições** da seguinte forma:

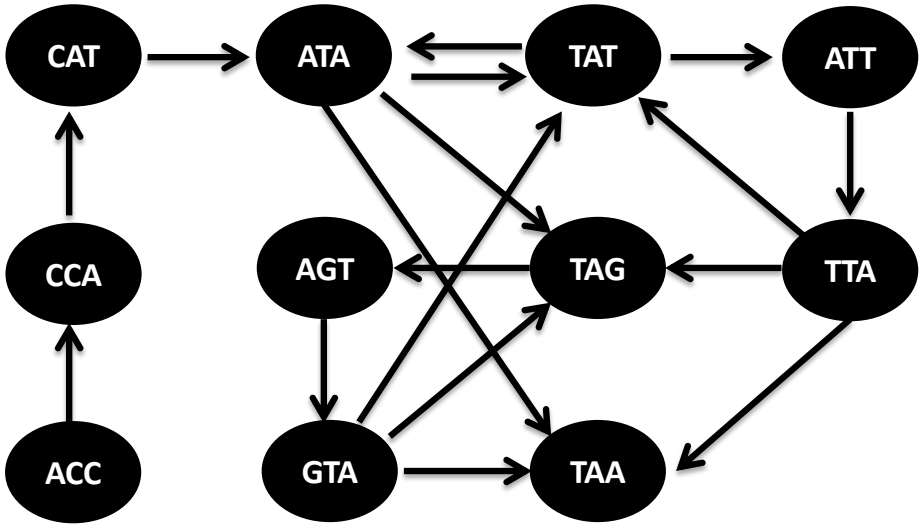
Nós correspondem às sequências dos diversos segmentos

Arcos são definidos entre nós A -> B, se sufixo_{k-1} (A) = prefixo_{k-1} (B)

Assim, reconstrução da string original corresponde a um **caminho no grafo de sobreposições**

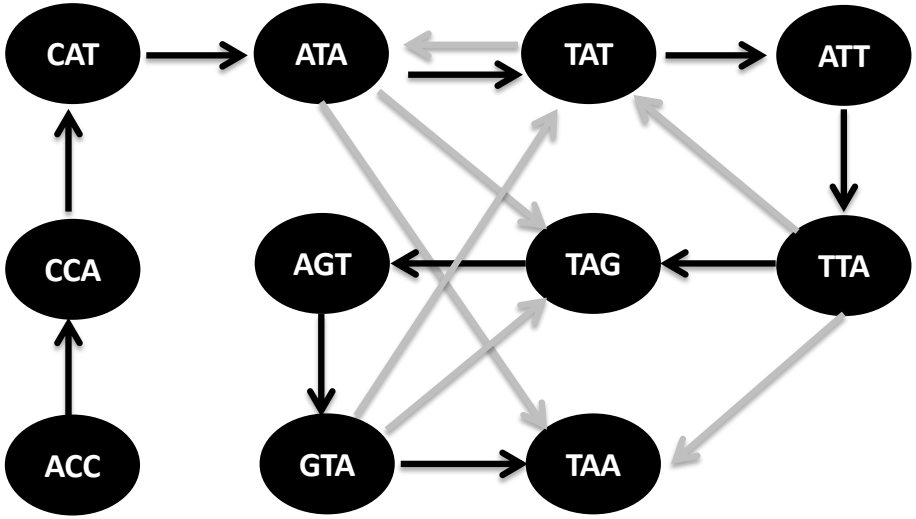
Necessário transformar o caminho na string resultante

Grafo de sobreposições



Grafo completo para o exemplo anterior

Caminho no grafo representando a solução anterior



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, frags):
        MyGraph.__init__(self, {})
        self.create_overlap_graph(frags)

    def create_overlap_graph(self, frags):
        ## add vertices
        ...

        ## add edges
        ...
```

```
def test2():
    frags = [”ACC", ”ATA", ”CAT", ”CCA", ”TAA”]
    ovgr = OverlapGraph(frags)
    ovgr.print_graph()

test2()
```

Implementando grafo de sobreposições

```
def create_overlap_graph(self, frags):
    ## add vertices
    for seq in frags:
        self.add_vertex(seq)

    ## add edges
    for seq in frags:
        suf = suffix(seq)
        for seq2 in frags:
            if prefix(seq2) == suf:
                self.add_edge(seq, seq2)
```

```
## auxiliary functions
def suffix (seq):
    return seq[1:]

def prefix (seq):
    return seq[:-1]
```

Definidas fora da classe

Qual o problema desta solução?

Implementando grafo de sobreposições - exemplo com repetições

```
def test3():
        frags = ["ATA", "ACC", "ATG", "ATT", "CAT”, "CAT", "CAT",, "CCA", "GGC",
        “TAA”, “TCA”, "TGG", "TTC", "TTT"]
        ovgr = OverlapGraph(frags)
        ovgr.print_graph()

test3()
```

Implementando grafo de sobreposições com repetições

```
def create_overlap_graph_with_reps (self, frags):
        idnum = 1
        for seq in frags:
                self.add_vertex(seq+ "-" + str(idnum))
                idnum = idnum + 1
        idnum = 1
        for seq in frags:
                suf = suffix(seq)
                for seq2 in frags:
                        if prefix(seq2) == suf:
                                for x in self.get_instances(seq2):
                                        self.add_edge(seq+ "-" + str(idnum), x)
                idnum = idnum + 1
```

```
def __init__(self, frags, reps = True):
        MyGraph.__init__(self, {})
        if reps:
                self.create_overlap_graph_with_reps (frags)
        else: self.create_overlap_graph(frags)
        self.reps = reps
```

```
def get_instances (self, seq):
        res = [ ]
        for k in self.graph.keys():
                if seq in k:
                        res.append(k)
        return res
```

```
def test3():
        frags = ["ATA", "ACC", "ATG", "ATT", "CAT”,
        "CAT", "CAT",, "CCA", "GGC", “TAA”, “TCA”,
        "TGG", "TTC", "TTT"]
        ovgr = OverlapGraph(frags, True)
        ovgr.print_graph()

test3()
```

Circuitos Hamiltonianos

Num grafo orientado, um **Circuito Hamiltoniano** é um **caminho que passa por todos os nós do grafo exatamente uma vez**

O **problema da reconstrução de uma string a partir da sua composição em k-mers** pode ser formulado como a **procura de um circuito Hamiltoniano no grafo de sobreposições** (com repetições) da forma definida anteriormente

A partir dum caminho Hamiltoniano a sequência que gerou os k-mers representados no grafo pode ser determinada facilmente !

Implementando caminhos Hamiltonianos

```
def check_if_valid_path(self, p):
    if p[0] not in self.graph.keys(): return False
    for i in range(1,len(p)):
        if p[i] not in self.graph.keys() or p[i] not in self.graph[p[i-1]]:
            return False
    return True

def check_if_hamiltonian_path (self, p):
    if not self.check_if_valid_path(p): return False
    …
```

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

Implementando caminhos Hamiltonianos

```
def check_if_hamiltonian_path(self, p):
    if not self.check_if_valid_path(p):
        return False
    to_visit = list(self.get_nodes())
    if len(p) != len(to_visit):
        return False
    for i in range(len(p)):
        if p[i] in to_visit: to_visit.remove(p[i])
        else: return False
    if not to_visit:
        return True
    else:
        return False
```

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

Implementando caminhos Hamiltonianos

```
def test4():
      frags = ["ATA",  "ACC", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA” , "GCA", "GGC",
"TAA", "TCA", "TGG", "TTC", "TTT"]
      ovgr = OverlapGraph(frags, True)
      path = [’ACC−2’, ’CCA−8’, ’CAT−5’, ’ATG−3’]
      print (ovgr.check_if_valid_path(path))
      print (ovgr.check_if_hamiltonian_path(path))
      path2 = [’ACC−2’, ’CCA−8’, ’CAT−5’, ’ATG−3’, ’TGG−13’, ’GGC−10’, ’GCA−9’, ’CAT−6’,
’ATT−4’, ’TTT−15’, ’TTC−14’, ’TCA−12’, ’CAT−7’, ’ATA−1’, ’TAA−11’]
      print (ovgr.check_if_valid_path(path2))
      print (ovgr.check_if_hamiltonian_path(path2))

test4()
```

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def seq_from_path (self, path):
    ...
```

Função para dar a sequência reconstruída dado um caminho no grafo

```
def get_seq (self, node):
    if node not in self.graph.keys(): return None
    if self.reps: return node.split("-")[0]
    else: return node
```

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def seq_from_path (self, path):
    if not self.check_if_hamiltonian_path(path):
        return None
    seq= self.get_seq(path[0])
    for i in range(1,len(path)):
        nxt = self.get_seq(path[i])
        seq += nxt[-1]
    return seq
```

Implementando caminhos Hamiltonianos: recuperação da sequência

```
def test4():
    frags = [ "ACC", "ATA", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA" , "GCA", "GGC",
"TAA", "TCA", "TGG", "TTC", "TTT"]
    ovgr = OverlapGraph(frags, True)
    path2 = ['ACC−2', 'CCA−8', 'CAT−5', 'ATG−3', 'TGG−13', 'GGC−10', 'GCA−9', 'CAT−6',
'ATT−4', 'TTT−15', 'TTC−14', 'TCA−12', 'CAT−7', 'ATA−1', 'TAA−11']
    print (ovgr.check_if_valid_path(path2))
    print (ovgr.check_if_hamiltonian_path(path2))
    print (ovgr.seq_from_path(path2))

test3()
```

Circuitos Hamiltonianos

É fácil verificar se um caminho é Hamiltoniano ... mas ...

O problema de procura de circuitos é um problema de elevada complexidade, categorizado como **NP-completo**, o que significa que não há algoritmos eficientes para determinar a solução ótima. Assim, quando o nº de nós do grafo aumenta, o problema torna-se impossível de resolver em tempo útil.

Uma alternativa óbvia, ainda que não escalável para grafos de grandes dimensões, é realizar uma **procura exaustiva**

Esta é baseada numa travessia em **profundidade** do grafo, com *backtracking* quando se atinge um nó a partir do qual não é possível adicionar um sucessor ainda não visitado nesse caminho

Implementando caminhos Hamiltonianos: procura exaustiva

```
def search_hamiltonian_path(self):
        for ke in self.graph.keys():
                p = self.search_hamiltonian_path_from_node(ke)
                if p != None:
                        return p
        return None

def search_hamiltonian_path_from_node (self, start):
        …
```

Funções para procura de caminhos *Hamiltonianos*: em todo o grafo e indicando o nó inicial
Adicionadas na classe **MyGraph**

Implementando caminhos Hamiltonianos: procura exaustiva

```
def search_hamiltonian_path_from_node(self, start):
    current = start
    visited = {start:0}
    path = [start]
    while len(path) < len(self.get_nodes()):
        nxt_index = visited[current]
        if len(self.graph[current]) > nxt_index:
            nxt_node = self.graph[current][nxt_index]
            visited[current] += 1
            if nxt_node not in path:
                path.append(nxt_node)
                visited[nxt_node] = 0
                current = nxt_node
        else:
            if len(path) > 1:
                rmv_node = path.pop()
                del visited[rmv_node]
                current = path[-1]
            else: return None
    return path
```

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

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

Caso em que nó é adicionado ao caminho

Backtracking: recuar para testar caminhos alternativos

Implementando caminhos Hamiltonianos: procura exaustiva

```
def test5():
    frags = [ "ACC", "ATA", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA", "GCA",
"GGC", "TAA", "TCA", "TGG", "TTC", "TTT"]
    ovgr = OverlapGraph(frags, True)

    path = ovgr.search_hamiltonian_path()
    print(path)
    print (ovgr.check_if_hamiltonian_path(path))
    print (ovgr.seq_from_path(path))

test5()
```

Implementando caminhos Hamiltonianos: fechando o ciclo

```
def test6():
    orig_sequence = "CAATCATGATGATGATC"
    frags = composition(3, orig_sequence)
    ovgr = OverlapGraph(frags, True)
    path = ovgr.search_hamiltonian_path()
    print (path)
    print (ovgr.seq_from_path(path))

test6()
```

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 ?

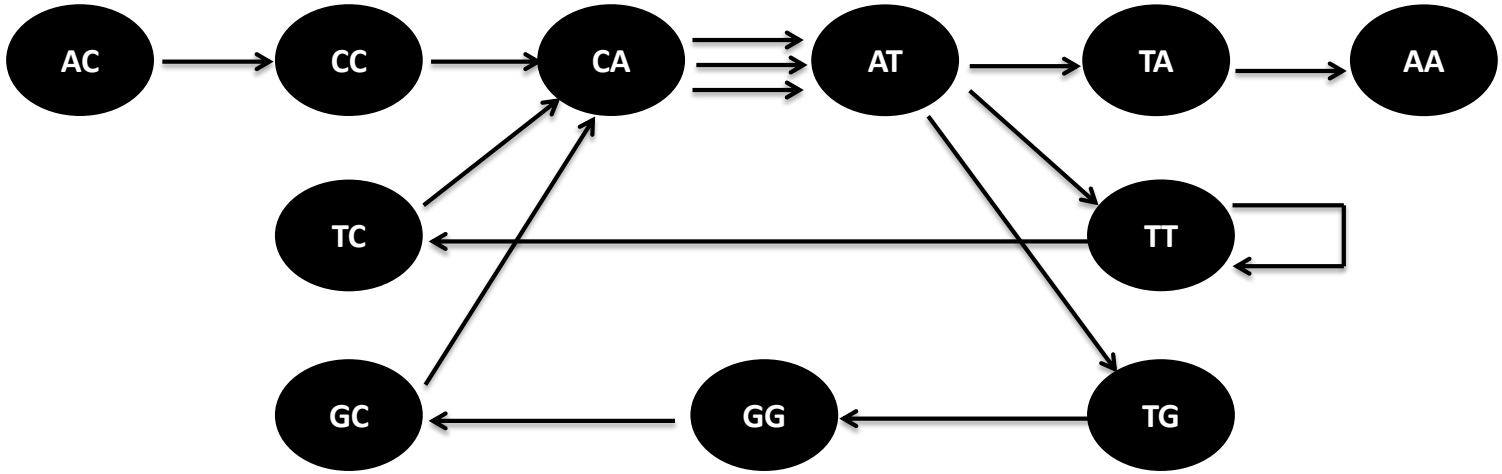
Representação alternativa: grafos de DeBruijn

Abordando problemas semelhantes, o matemático **DeBruijn** propôs uma representação distinta do problema ainda usando grafos

A ideia passa por representar os fragmentos (k-mers) não como nós do grafo mas antes como **arcos**, sendo os **nós** sequências de tamanho k-1 correspondendo a **prefixos/ sufixos** destes fragmentos

O exemplo abaixo ilustra a abordagem para o conjunto de fragmentos anterior:

Grafos de DeBruijn



O exemplo ilustra a abordagem para o conjunto de fragmentos anterior:

Implementando grafo de DeBruijn

```
from MyGraph import MyGraph

class DeBruijnGraph (MyGraph):

    def __init__(self, frags):
        MyGraph.__init__(self, {})
        self.create_deBruijn_graph(frags)

    def add_edge(self, o, d):
        if o not in self.graph.keys():
            self.add_vertex(o)
        if d not in self.graph.keya():
            self.add_vertex(d)
        self.graph[o].append(d)

    …
```

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 create_deBruijn_graph(self, frags):
    ...
```

```
def test1():
    frags = [ "ACC", "ATA", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA",
"GCA", "GGC", "TAA", "TCA", "TGG", "TTC", "TTT"]

    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()

test1()
```

Implementando grafo de DeBruijn

```
def create_deBruijn_graph(self, frags):
    for seq in frags:
        suf = suffix(seq)
        self.add_vertex(suf)
        pref = prefix(seq)
        self.add_vertex(pref)
        self.add_edge(pref, suf)
```

```
def test1():
    frags = [ "ACC", "ATA", "ATG", "ATT", "CAT", "CAT", "CAT", "CCA",
"GCA", "GGC", "TAA", "TCA", "TGG", "TTC", "TTT"]

    dbgr = DeBruijnGraph(frags)
    dbgr.print_graph()

test6()
```

Circuitos Eulerianos

Num grafo orientado, um **Circuito Euleriano** é um **caminho que passa por todos os arcos do grafo exatamente uma vez**

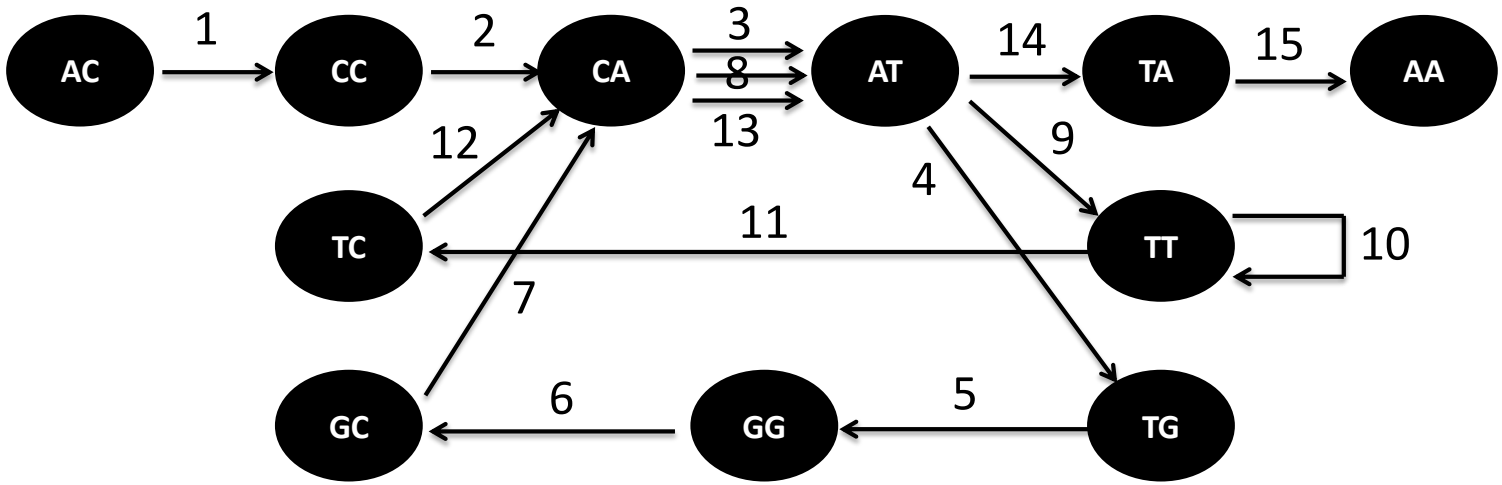
Da mesma forma, um **Ciclo Euleriano** passa por todos os arcos do grafo exatamente uma vez, regressando ao nó de partida

O **problema da reconstrução de uma string a partir da sua composição em k-mers** pode ser formulado como a **procura de um circuito Euleriano no grafo de DeBruijn** anterior

Temos, assim, duas formas diferentes, com dois grafos diferentes de resolver o mesmo problema. Ganhamos alguma coisa com isso?

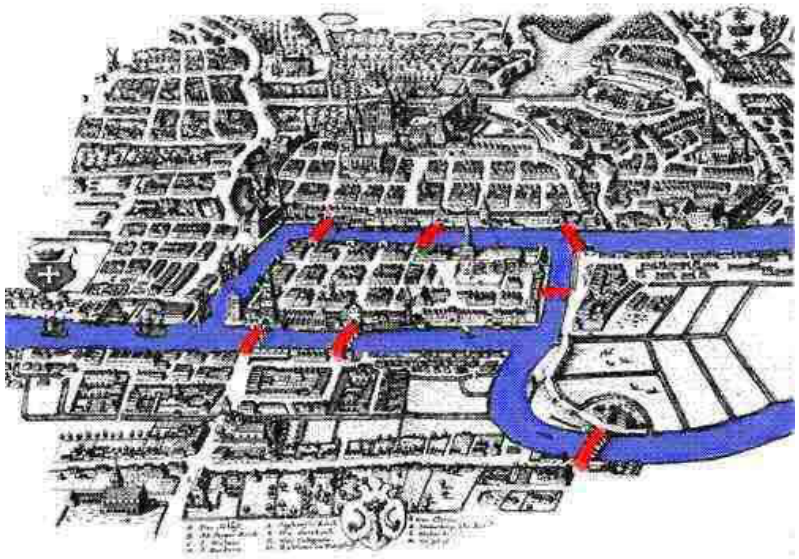
Para responder a esta questão temos que perceber como podemos identificar circuitos/ ciclos Eulerianos em grafos e qual a complexidade destes algoritmos

Circuitos Eulerianos para problema de reconstrução: exemplo



O início da teoria dos grafos

Descobrir um caminho que passe 1 vez em cada ponte,
Leonhard Euler, 1735



pontes de Königsberg

Teorema de Euler

Um grafo orientado é **balanceado** se para cada vértice o nº de arcos que "chegam" é igual ao nº de arcos que "partem":

grau entrada (v)= grau saída(v)

Teorema: *Um grafo conectado tem um ciclo Euleriano (é chamado grafo Euleriano) sse cada um dos seus vértices é balanceado.*

Num grafo **conectado**, todos os pares de nós estão ligados entre si por caminhos

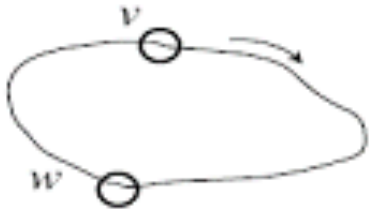
Será que há um teorema semelhante para grafos não orientados ?

Algoritmo para a construção de ciclos Eulerianos

Passo 1

Começar com um vértice *v* e formar um ciclo arbitrário usando arcos ainda não visitados até atingir um "beco sem saída", i.e. um nó sem sucessores atingíveis por arcos não usados.

Como o grafo é Euleriano paramos necessariamente em *v*.

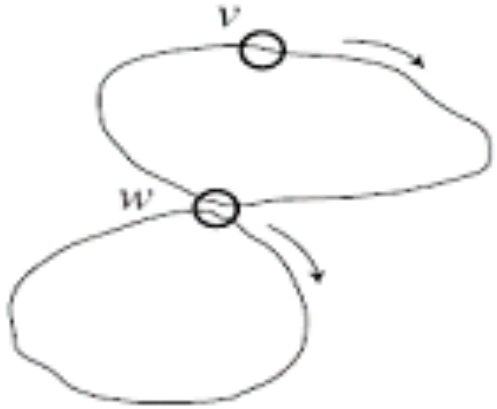


(a)

Algoritmo para Construção de um ciclo Euleriano (cont.)

Passo 2

Se ciclo do passo 1 ainda não é Euleriano (i.e. não contém todos os arcos), contém pelo menos um vértice *w*, com arcos não usados. Repetir o passo 1, usando *w* como ponto de partida. Este passo terminará em *w.*

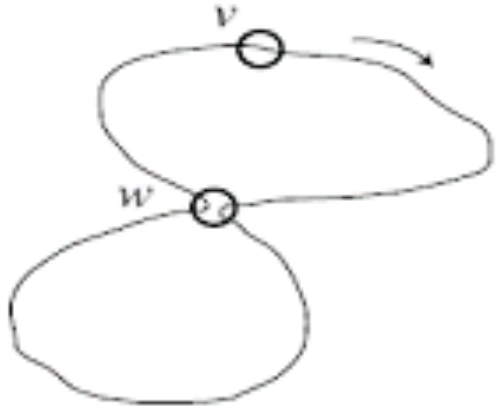


(b)

Algoritmo para Construção de um ciclo Euleriano (cont.)

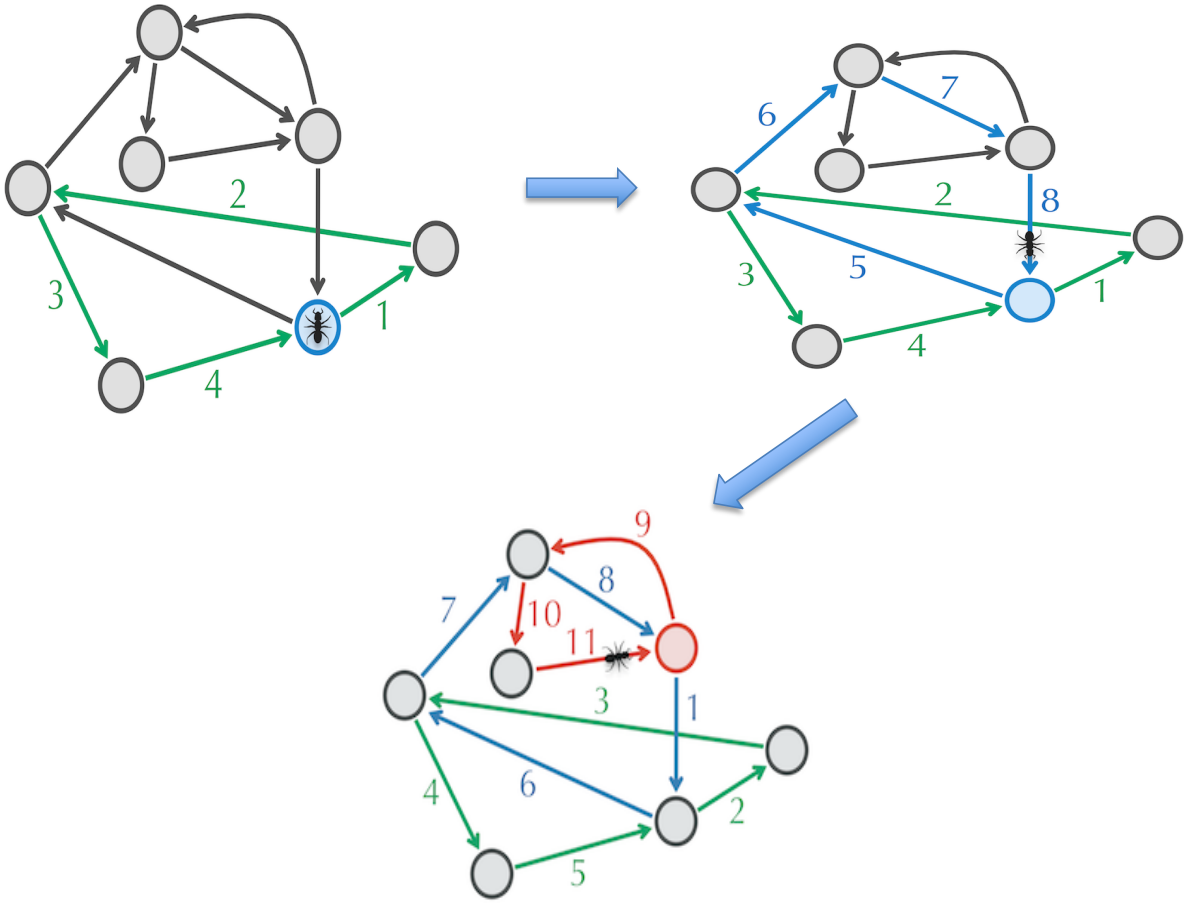
Passo 3

Combinar ciclos dos passos 1 e 2 num único. Iterar passo 2 até “cobrir” todos os arcos



(c)

Exemplo



Teorema Euler: Extensão

Teorema: *Um grafo conectado tem um* ***caminho (circuito) Euleriano*** *sse contém no máximo dois vértices semi-balanceados (diferença entre grau de entrada e saída é 1 e -1 respetivamente) e todos os outros são balanceados.*

Algoritmo*: (a) unir dois vértices semi-balanceados – passamos a ter grafo euleriano; (b) determinar ciclo euleriano no novo grafo; (c) remover do ciclo o arco inserido*

E no caso dos grafos não orientados ?

Circuitos Eulerianos

O problema de procura de circuitos Eulerianos é de **complexidade bastante inferior** ao seu homólogo para circuitos Hamiltonianos, podendo ser resolvido para grafos de grandes dimensões usando o algoritmo anterior (uma implementação eficiente garante tempos lineares)

Os programas de montagem de genomas só recentemente começaram a usar esta estratégia; os primeiros programas (e.g. os usados no projeto do Genoma Humano) usavam grafos de sobreposição e caminhos Hamiltonianos

Atualmente, muitos dos programas para montagem de genomas a partir de dados de sequenciação são baseados em grafos de DeBruijn e circuitos Eulerianos

Implementando ciclos Eulerianos
(classe MyGraph)

```
def check_balanced_node(self, node):
        return self.in_degree(node) == self.out_degree(node)

def check_balanced_graph(self):
        for n in self.graph.keys():
                if not self.check_balanced_node(n): return False
        return True
```

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

```
def eulerian_cycle(self):
        if not self.check_balanced_graph():
                return None
        edges_visit = list(self.get_edges())
        res = [ ]
        …
        return res
```

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