Procura de padrões em sequências

Organização de padrões em árvores - *tries*

Árvores n-árias que permitem organizar (um ou) vários padrões a procurar numa sequência

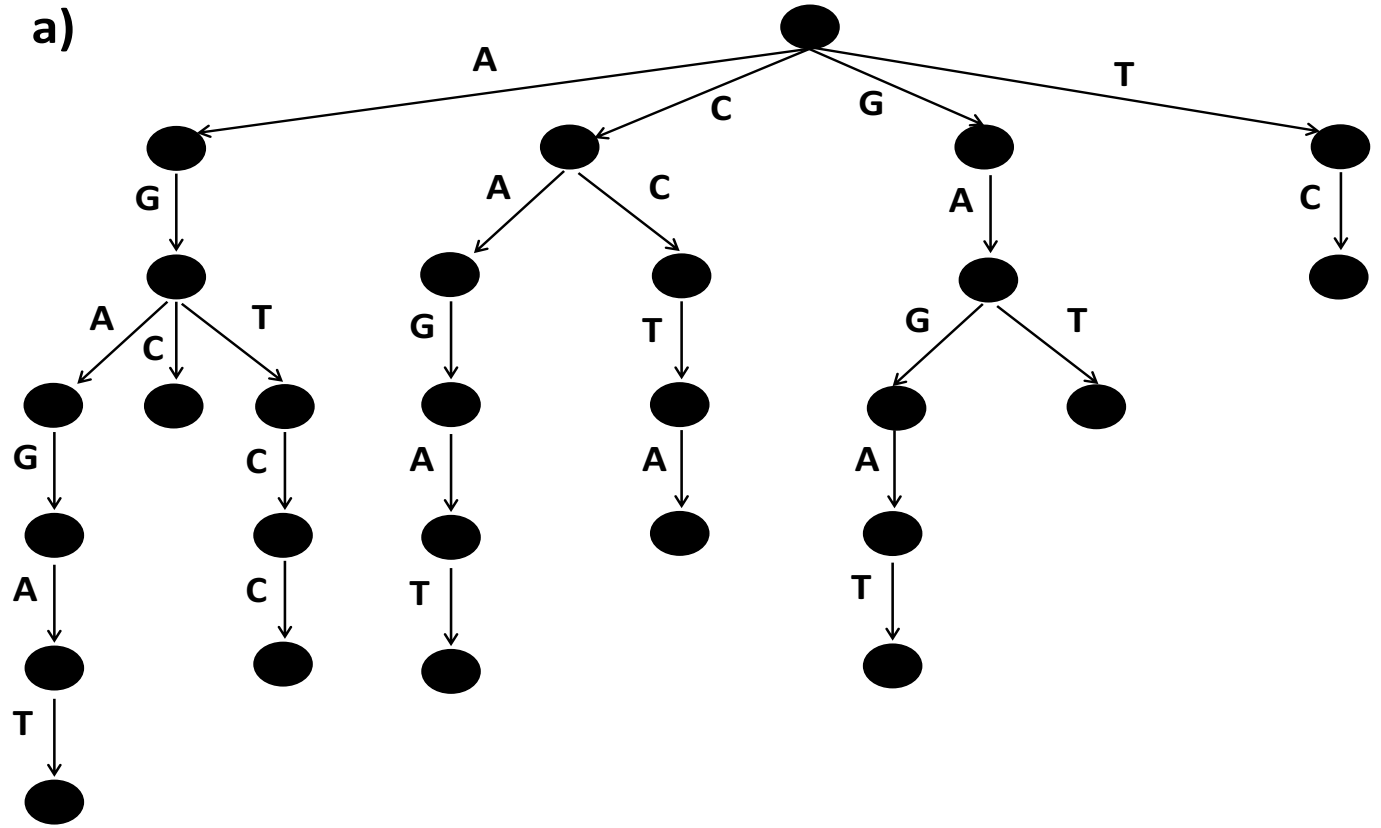
Cada arco é associado a um dos símbolos do alfabeto, sendo que cada arco que sai de um nodo é associado a um símbolo distinto

Árvore tem uma folha por cada padrão

Cada padrão pode ser construído juntando os símbolos da raiz até uma das folhas

Exemplo de uma *trie*

a)



b) Padrões representados

AGAGAT
AGC
AGTCC
CAGAT
CCTA
GAGAT
GAT
TC

Construção de uma *trie* a partir de padrões

A construção da trie faz-se iniciando com uma árvore apenas com raiz e adicionando cada padrão iterativamente

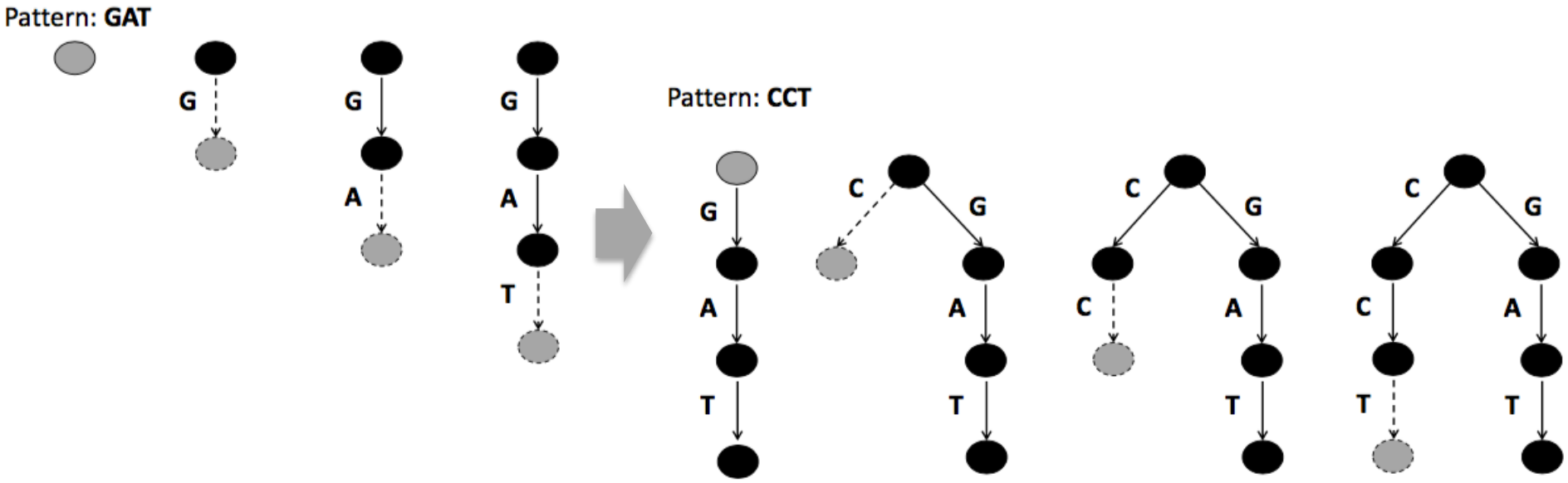
Ao adicionar cada padrão, definimos o nó atual como a raiz da árvore, e lemos o padrão do início para o final; para cada posição fazemos:

- Se o símbolo lido não existe nos arcos que saem do nó atual, criamos um novo nó e ligamos o nó atual ao novo nó; o nó atual passará a ser o novo nó
- Se o arco existe, “descemos” por esse arco e o nó atual passará a ser o nó destino deste arco

Construção de uma *trie* a partir de padrões

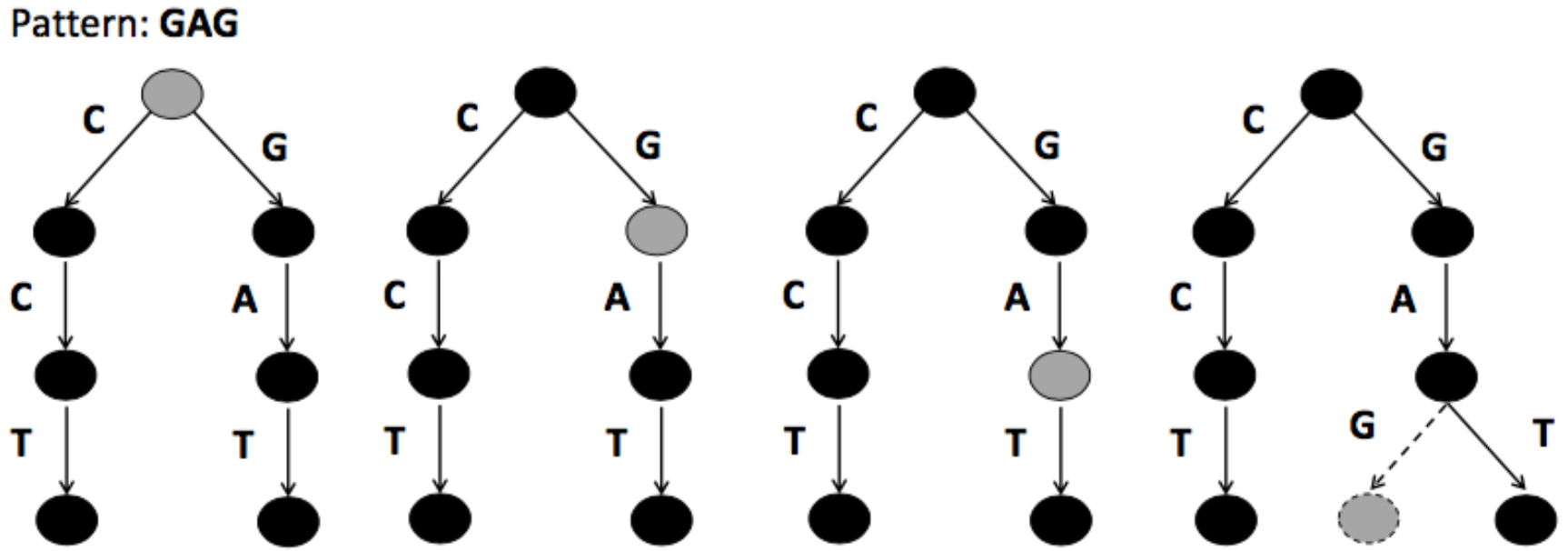
Passo 1:

Passo 2:



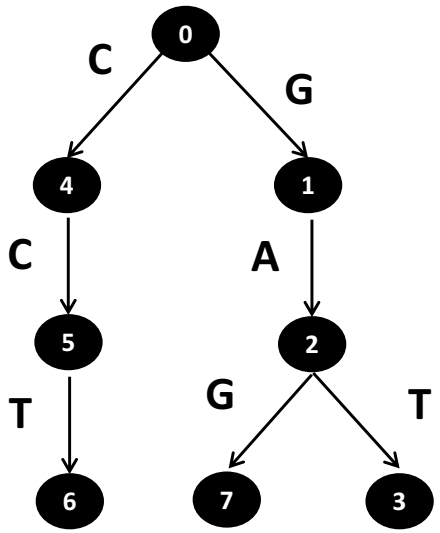
Construção de uma *trie* a partir de padrões

Passo 3:



Implementando tries

a)



b)

Origem

Key (node)	Value (Edges)
0	{ “C”: 4, “G”: 1 }
1	{ “A”: 2 }
2	{ “G”: 7, “T”: 3 }
3	{ }
4	{ “C”: 5 }
5	{ “T”: 6 }
6	{ }
7	{ }

8 nós indica 8 entradas

Destinos

OS dicionários vazios indicam se é nó ou folha, sendo os fazios folha pq não sai de lá nada.

Árvore guardada como **dicionário**: nº do nó -> arcos, em que os arcos de um nó são representados por um outro dicionário: símbolo -> nº de nó destino

Nós são numerados à medida que são criados (*self.num* guarda nº do último criado)

Implementando tries

```
class Trie:

        def __init__(self):
                self.nodes = { 0:{} } # root node
                self.num = 0

        def print_trie(self):
                for k in self.nodes.keys():
                        print (k, "->" , self.nodes[k])

        def add_node(self, origin, symbol):
                self.num += 1
                self.nodes[origin][symbol] = self.num
                self.nodes[self.num] = {}
...
```

Implementando tries

```
class Trie:

    def add_pattern(self, p):
        ...


    def trie_from_patterns(self, pats):
        ...
```

Adiciona padrão à trie

Adiciona conjunto de padrões à trie

Implementando tries

```
class Trie:

    def add_pattern(self, p):
        pos = 0
        node = 0
        while pos < len(p):
            if p[pos] not in self.nodes[node].keys() :
                self.add_node(node, p[pos])
            node = self.nodes[node][p[pos]]
            pos += 1

    def trie_from_patterns(self, pats):
        for p in pats:
            self.add_pattern(p)
```

```
def test():
    patterns = ["GAT", "CCT", "GAG"]
    t = Trie()
    t.trie_from_patterns(patterns)
    t.print_trie()

test()
```

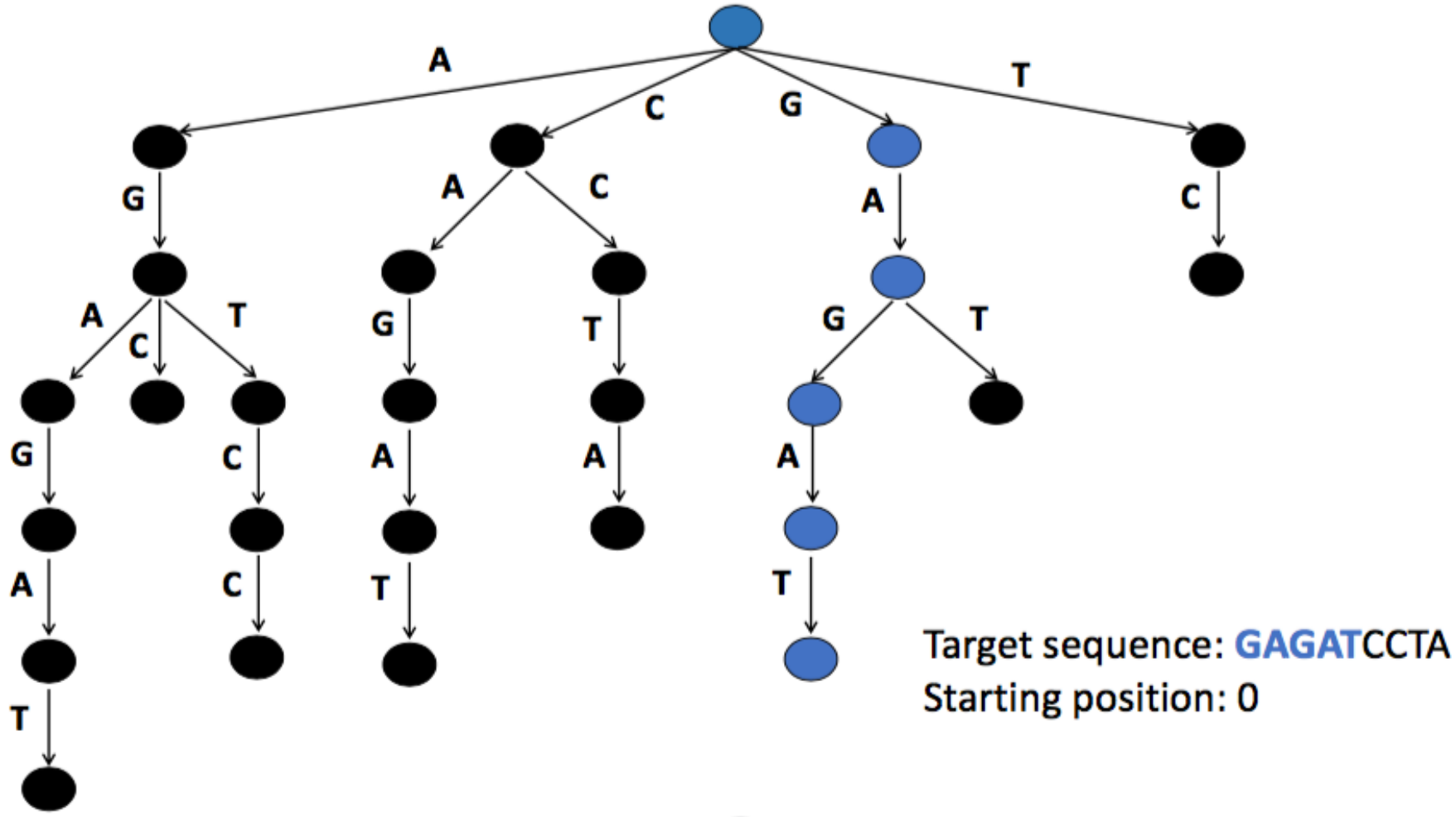
Identificação de matches usando *tries*

Dada uma árvore criada a partir de um conjunto de padrões, pode-se facilmente **procurar a ocorrência de um desses padrões como prefixo de uma sequência**

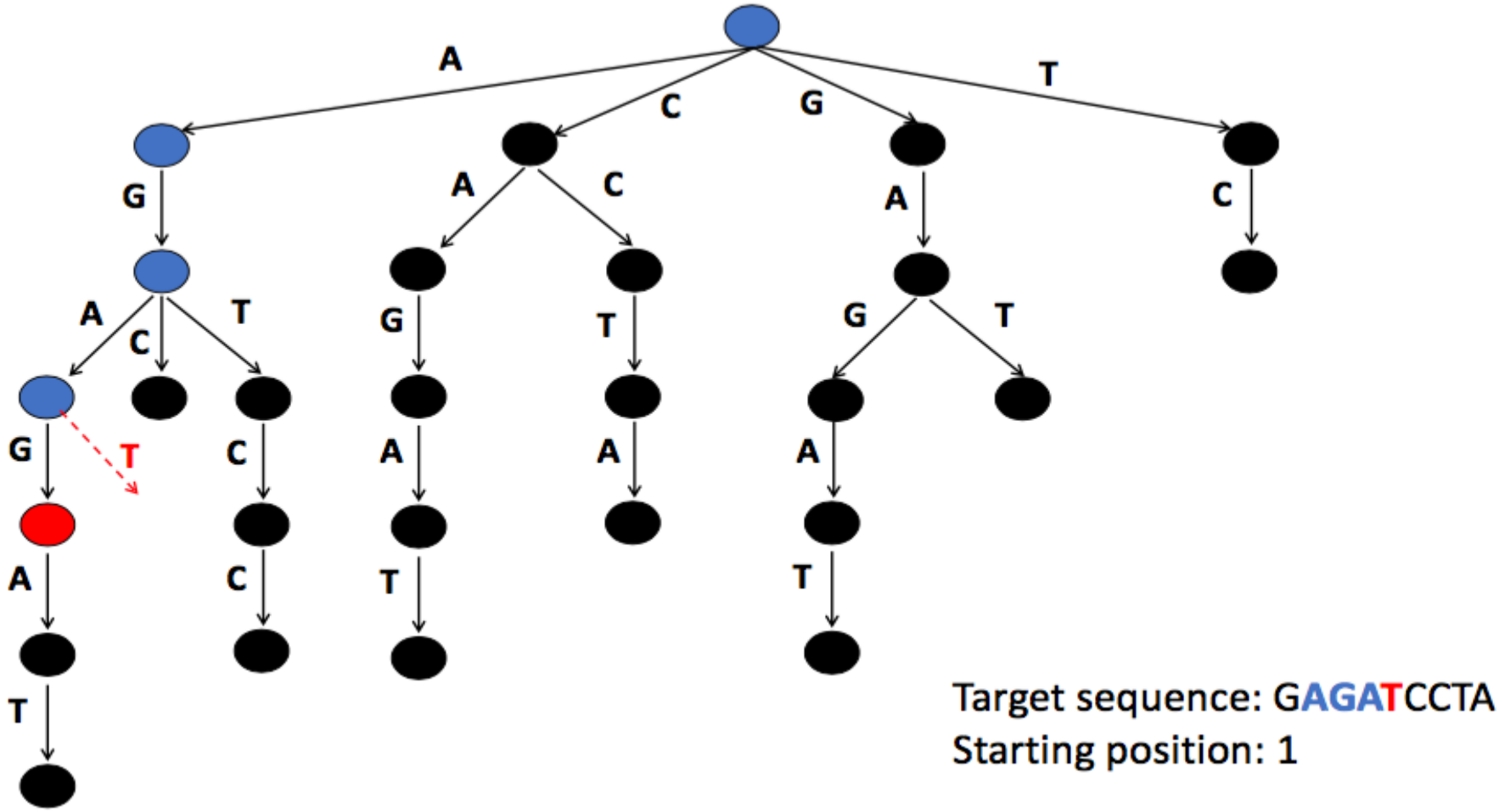
Neste caso, basta percorrer a árvore (saindo do nó raiz) e seguir os arcos correspondentes; se se atingir uma folha, o padrão correspondente a esta folha é prefixo da sequência

Para identificar matches dos padrões em **toda a sequência**, faz-se um processo iterativo onde se vai fazendo o match da sequência e depois se **remove o primeiro** símbolo desta, repetindo o processo; assim, o processo anterior é repetido para **todos os sufixos** da sequência

Identificacão de matches usando *tries*: exemplos



Identificação de matches usando *tries*: exemplos



Implementando procura de padrões com tries

```
def prefix_trie_match(self, text):
        ...


def trie_matches(self, text):
        ...
```

Procura de padrões como prefixos da sequência *text*

Procura de padrões na sequência *text*

Implementando procura de padrões com tries

```
def prefix_trie_match(self, text):
        pos = 0
        match = ""
        node = 0
        while pos < len(text):
                if text[pos] in self.nodes[node].keys() :
                        node = self.nodes[node][text[pos]]
                        match += text[pos]
                        if self.nodes[node] == {}: return match
                        else: pos += 1
                else: return None
        return None

def trie_matches(self, text):
        res = []
        for i in range(len(text)):
                m = self.prefix_trie_match(text[i:])
                if m != None: res.append((i,m))
        return res
```

```
def test2():
        patterns = ["AGAGAT", "AGC", "AGTCC", "CAGAT", "CCTA",
"GAGAT", "GAT", "TC"]
        t = Trie()
        t.trie_from_patterns(patterns)
        print (t.prefix_trie_match("GAGATCCTA"))
        print (t.trie_matches("GAGATCCTA") )

test()
```

Tries: discussão

As tries permitem procurar um conjunto alargado de padrões sobre a mesma sequência

Alguns problemas:

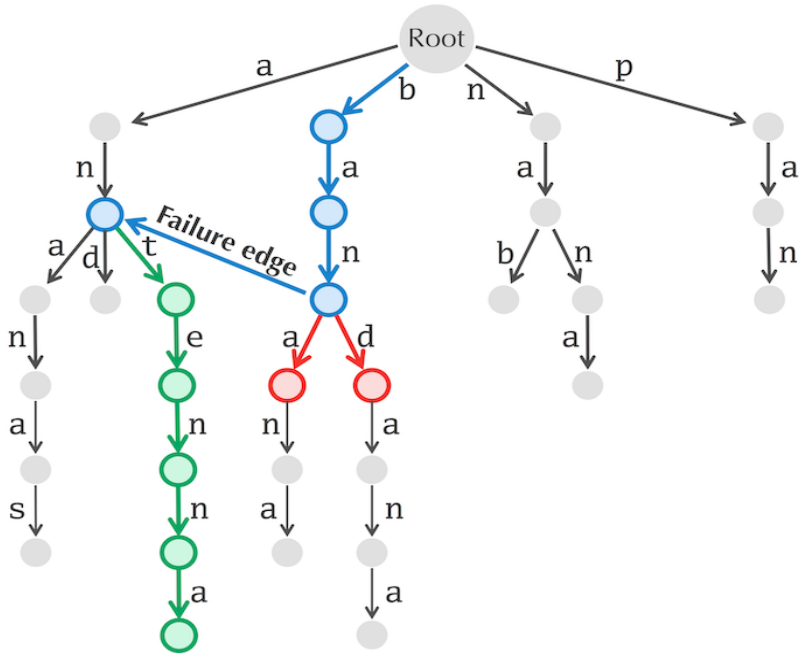
- Em cada iteração, a procura dos padrões é recomeçada para a nova sequência "amputada" do seu primeiro símbolo não se aproveitando informação recolhida no processo de match de cada "sufixo" (e.g. partes de padrões ...)
- Obrigam a que o nº de arcos na árvore seja grande, se tivermos um padrão de elevada dimensão e/ ou em grande número

Autómatos vs tries

Os autómatos podem ser construídos, tal como as *tries* anteriores, para várias sequências

AFs têm a vantagem de, quando há mismatch, não recuarem para a raiz (estado inicial) e assim serem mais eficientes

AFs podem ser encarados como tries com arcos a ligar nós em diferentes ramos para situações de mismatch (ver exemplo)



Exemplo: procura na sequência: “bantenna”

Árvores de sufixos

As árvores de sufixos permitem, ao contrário do que vimos até agora, fazer um **pré-processamento da sequência** na qual se quer procurar o padrão (ou padrões)

Depois de construída uma árvore de sufixos que representa a sequência pode procurar-se o padrão em tempo linear em relação ao tamanho do padrão

Especialmente relevante se quisermos procurar vários padrões na mesma sequência

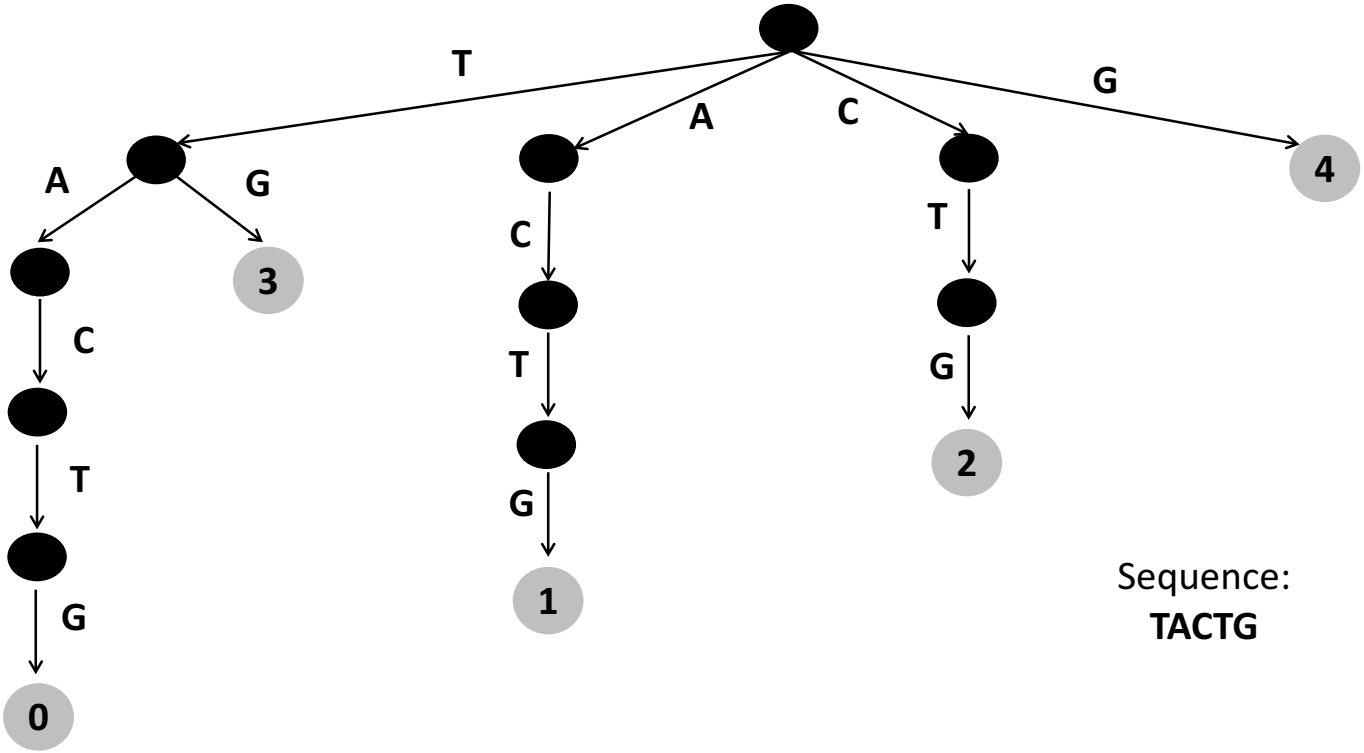
Árvores de sufixos são casos especiais de *tries*

Árvores de sufixos: definição

A árvore ***T = (V,E)*** pode ser considerada uma **árvore de sufixos da sequência *s***, dado o alfabeto ***A*** ao qual os símbolos de s pertencem, se:
- A árvore ***A*** tem um nº de folhas igual ao tamanho de ***s***
- As folhas têm como etiqueta *0, …, tamanho(s)-1*
- Cada ramo em ***E*** tem como etiqueta um valor de *A*
- Cada **sufixo de *s*** é representado por uma **folha** em *V*
- A concatenação dos símbolos no caminho da raiz até cada folha corresponde a um sufixo (etiqueta da folha corresponde à posição inicial do sufixo)
- Todos os ramos que saem de cada nó em *V* têm símbolos distintos

Ou seja, **árvores de sufixos são tries construídas com todos os sufixos de uma sequência, com folhas numeradas**

Árvore de sufixos: exemplo



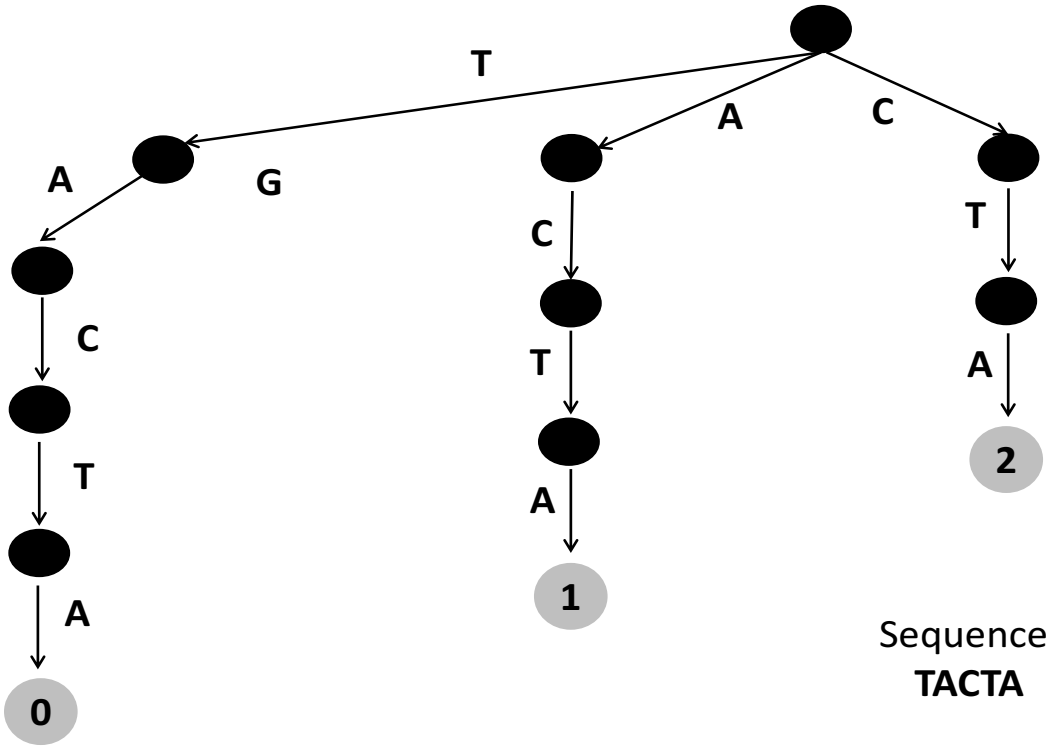
Exemplo:

Folha: 1
Caminho: ACTG
Sufixo iniciado em 1

Sequence:
TACTG

Árvore de sufixos: exemplo

Existem árvores de sufixos para todas as sequências ?



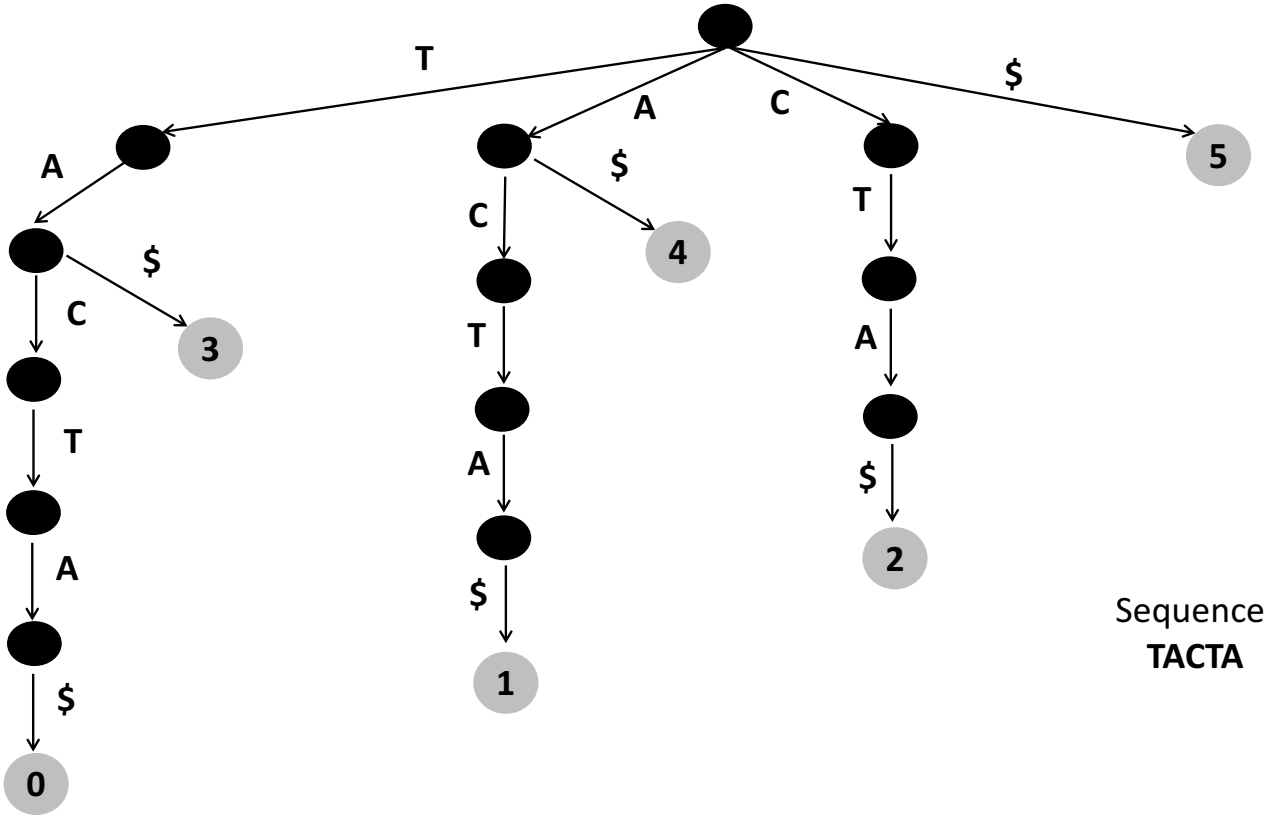
Sequence:
TACTA

Folhas 3 e 4 não existem:

Correspondem a prefixos de outros sufixos mais longos

Árvore de sufixos com todas as folhas ...

Solução: introduzir símbolo $ (único) no final da string …



Com esta representação, nenhum sufixo pode ser prefixo de outro …

Sequence:
TACTA

Construção da árvore de sufixos: algoritmo

Devem considerar-se todos os possíveis sufixos da sequência começando pela própria sequência e retirando iterativamente um símbolo no início

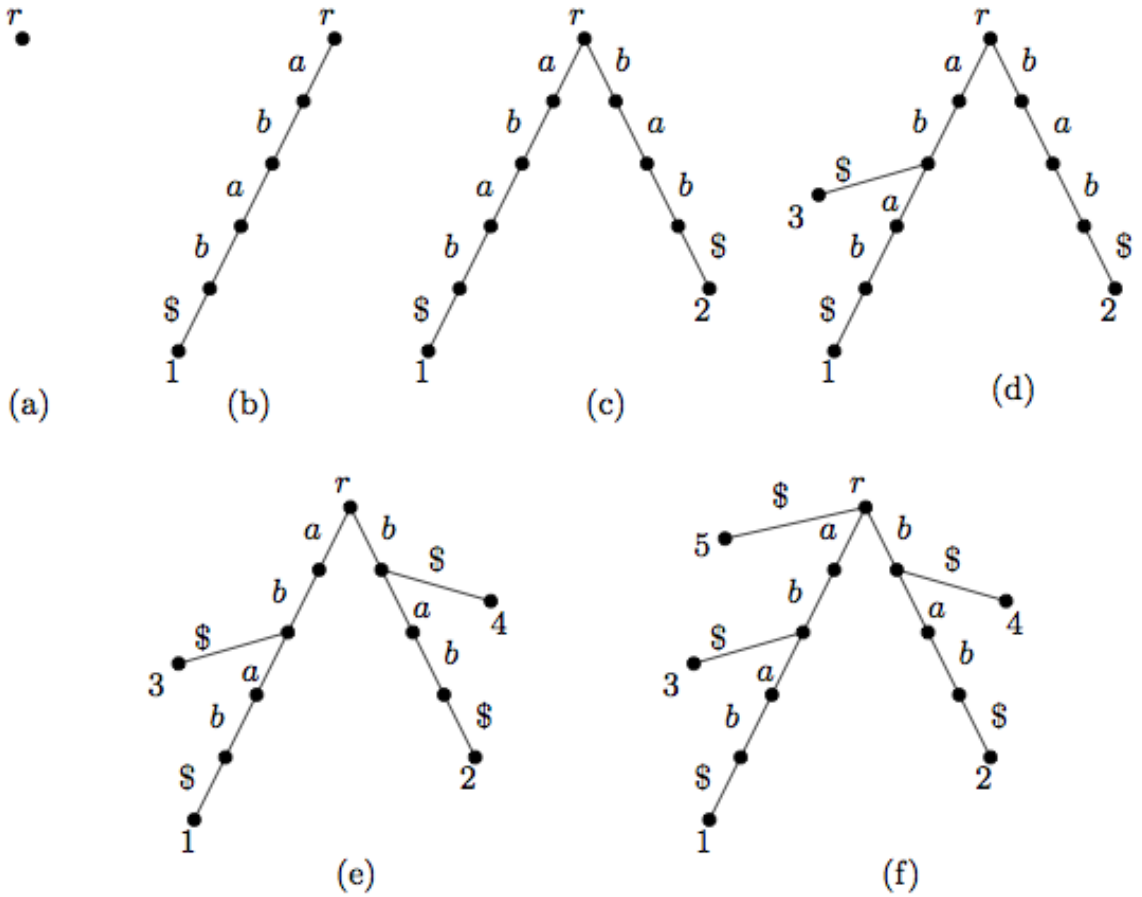
Os sufixos serão introduzidos na árvore um a um

Na introdução de cada sufixo, percorre-se a árvore seguindo os ramos correspondentes aos símbolos do sufixo, enquanto tal for possível

Quando se atingir um nó onde o ramo respectivo não exista este será criado

Serão criados abaixo deste ramo todos os ramos correspondentes ao sufixo até se atingir a folha respetiva

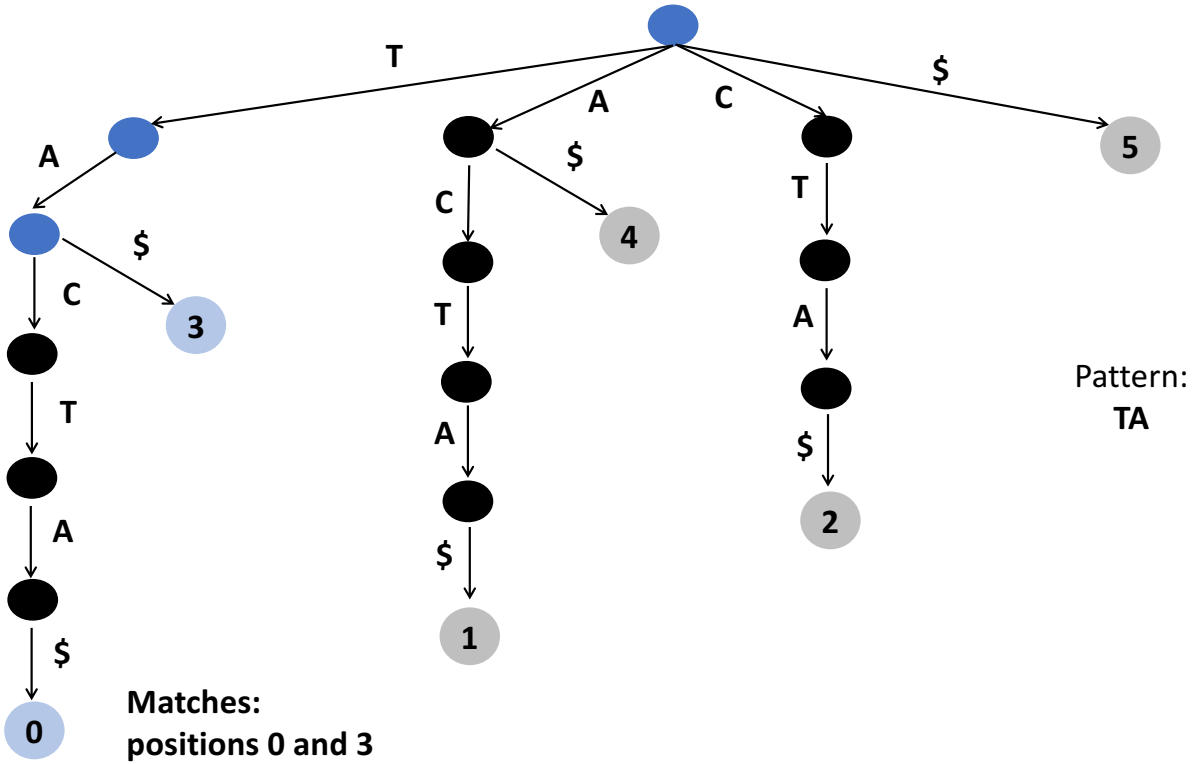
Construção da árvore: exemplo



Seq:
abab$

Árvore de sufixos: procura de padrões

Exemplo: padrão **TA**
em **TACTA$**
ocorre nas posições 0 e 3



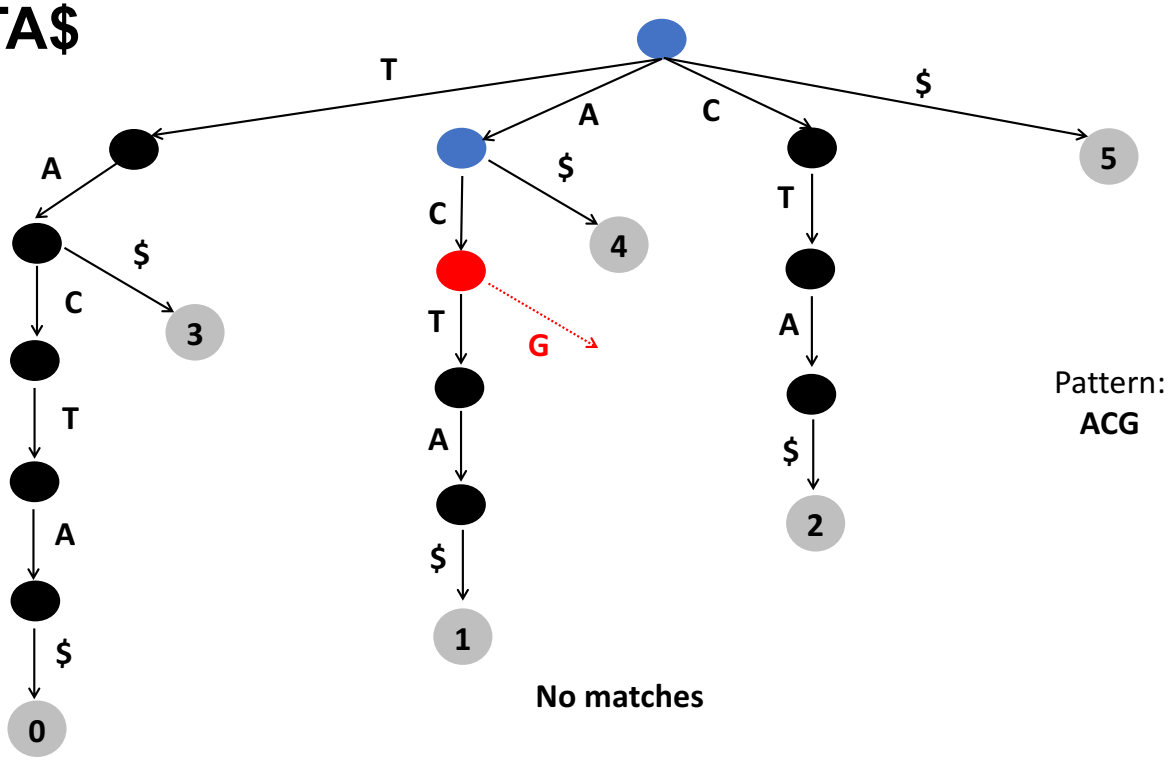
Procurar nó correspondente ao caminho representado pelo padrão:
- **Se nó existe, ocorrências do padrão são todas as folhas abaixo deste nó**
- Se nó não existe, padrão não ocorre

Árvore de sufixos: procura de padrões

Exemplo: padrão

ACG não ocorre em

TACTA$



Procurar nó correspondente ao caminho representado pelo padrão:

- Se nó existe, ocorrências do padrão são todas as folhas abaixo deste nó
- **Se nó não existe, padrão não ocorre**

Implementando árvores de sufixos

```
class SuffixTree:

    def __init__(self):
        self.nodes = { 0:(-1,{}) } # root node
        self.num = 0

    def print_tree(self):
        for k in self.nodes.keys():
            if self.nodes[k][0] < 0:
                print (k, "->", self.nodes[k][1])
            else:
                print (k, ":", self.nodes[k][0])

    def add_node(self, origin, symbol, leafnum = -1):
        self.num += 1
        self.nodes[origin][1][symbol] = self.num
        self.nodes[self.num] = (leafnum,{})
```

Estrutura da Trie é adaptada: para cada nó existe um tuplo onde o 1º elemento é o nº do sufixo (para folhas) ou -1 (se não é folha); o 2º elemento é o dicionário anterior da Trie (símbolo -> nº de nó destino)

Implementando árvores de sufixos

```
def add_suffix(self, p, sufnum):
        ...



def suffix_tree_from_seq(self, text):
        ...
```

Função para adicionar sufixo
Adaptar de *add_pattern* da Trie

Função que cria a árvore de sufixos
Adiciona um sufixo em cada iteraçção usando a anterior

```
def test1():
        seq = "TACTA"
        st = SuffixTree()
        st.suffix_tree_from_seq(seq)
        st.print_tree()
test1()
```

Implementando árvores de sufixos

```
def add_suffix(self, p, sufnum):
        pos = 0
        node = 0
        while pos < len(p):
                if p[pos] not in self.nodes[node][1].keys():
                        if pos == len(p)-1:
                                self.add_node(node, p[pos], sufnum)
                        else:
                                self.add_node(node, p[pos])
                node = self.nodes[node][1][p[pos]]
                pos += 1

def suffix_tree_from_seq(self, text):
        t = text+"$"
        for i in range(len(t)):
                self.add_suffix(t[i:], i)
```

```
def test1():
        seq = "TACTA"
        st = SuffixTree()
        st.suffix_tree_from_seq(seq)
        st.print_tree()
test1()
```

Implementando árvores de sufixos

```
def find_pattern(self, pattern):
        ...



def get_leafes_below(self, node):
        res = []
        ...
        return res
```

Função para procura de padrões usando a *Trie*

Usa função auxiliar para colecionar todas as folhas abaixo de um dado nó

```
def test2():
        seq = "TACTA"
        st = SuffixTree()
        st.suffix_tree_from_seq(seq)
        print st.find_pattern("TA")
test2()
```

Implementando árvores de sufixos

```
def find_pattern(self, pattern):
        pos = 0
        node = 0
        for pos in range(len(pattern)):
                if pattern[pos] in self.nodes[node][1].keys():
                        node = self.nodes[node][1][pattern[pos]]
                else: return None
        return self.get_leafes_below(node)


def get_leafes_below(self, node):
        res = []
        if self.nodes[node][0] >=0:
                res.append(self.nodes[node][0])
        else:
                for k in self.nodes[node][1].keys():
                        newnode = self.nodes[node][1][k]
                        leafes = self.get_leafes_below(newnode)
                        res.extend(leafes)
        return res
```

Função para procura de padrões usando a *Trie*

Usa função auxiliar para colecionar todas as folhas abaixo de um dado nó

```
def test2():
        seq = "TACTA"
        st = SuffixTree()
        st.suffix_tree_from_seq(seq)
        print st.find_pattern("TA")
test2()
```

Árvores de sufixos: compactação

Problema das árvores de sufixos: podem ser demasiado grandes se sequência for grande (e.g. um genoma humano ...)

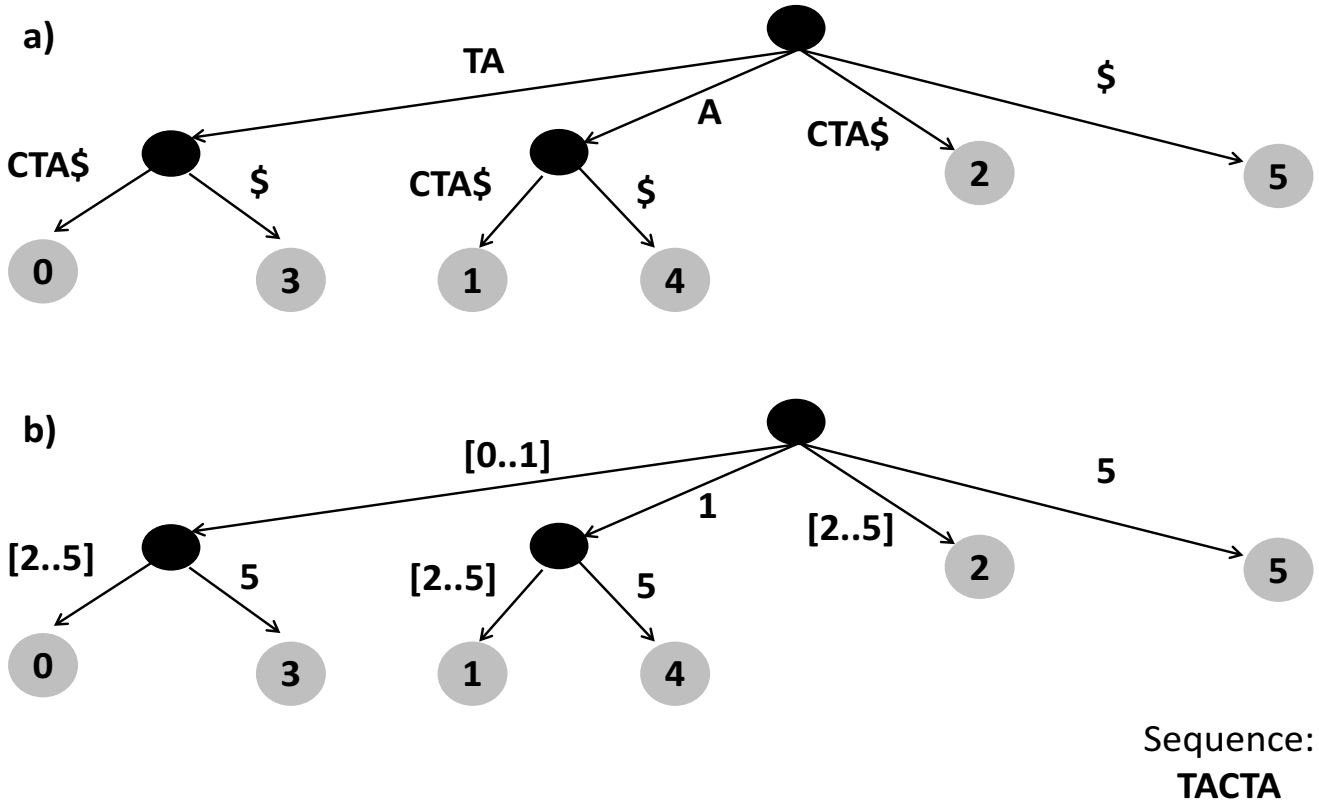
Existem representações de **árvores compactas** que reduzem o tamanho da árvore e o tempo de procura (e.g. compactando caminhos sem bifurcações num só ramo)

Os algoritmos de construção de árvores de sufixos compactas podem ser corridos em **O(n log n)**, onde n é o tamanho da sequência

Logo, o problema de procura de um padrão poderá ser resolvido em **O(n log n + m. |A| + k)**, onde m é o tamanho do padrão e k o nº de ocorrências do mesmo

Especialmente úteis para procurar vários padrões na mesma sequência

Árvore compacta: exemplo



Com as sequências dos sufixos

Com índices

Sequence:
TACTA

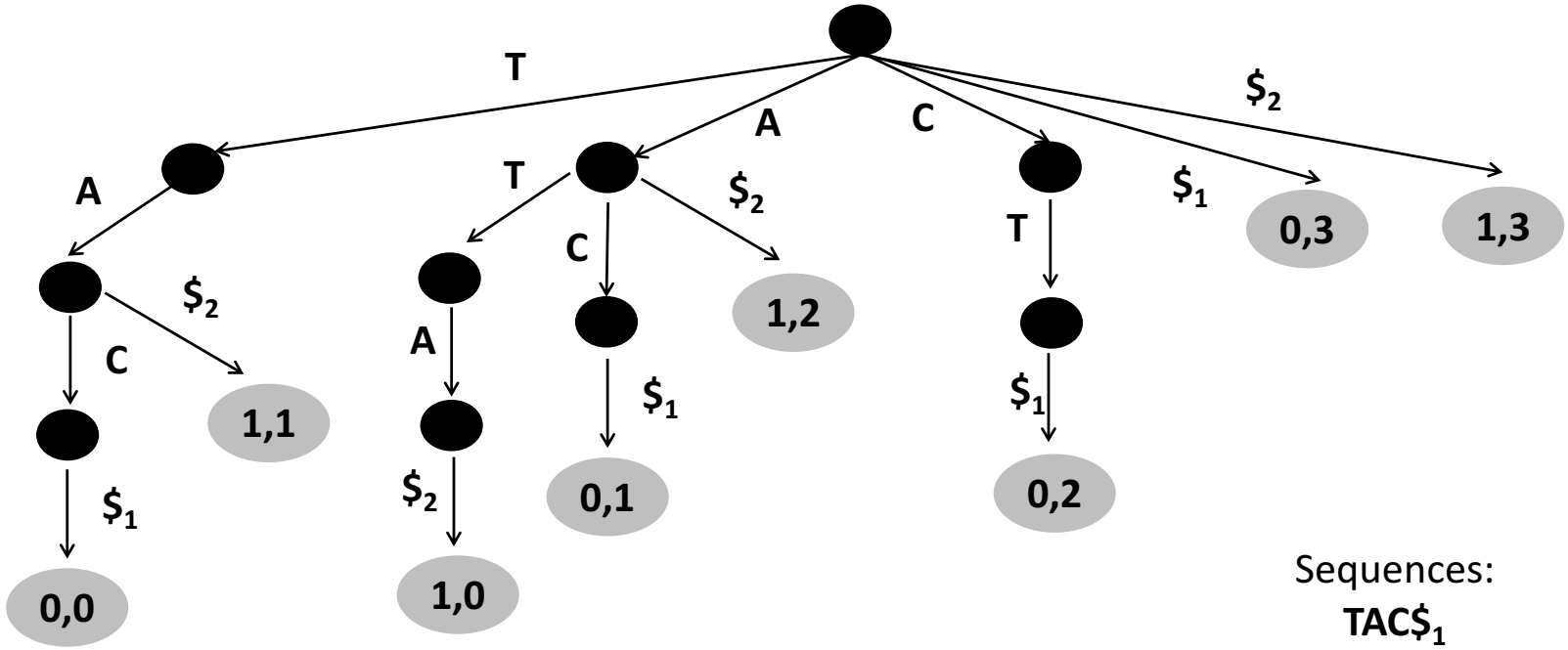
Árvores de sufixos: outras aplicações

Podem ser criadas árvores que representem várias sequências permitindo

- Descobrir quais contêm uma dada substring
- Descobrir a substring mais longa, partilhada por um conjunto de strings
- Computação eficiente de overlaps entre pares de sequências

As árvores de sufixos podem ainda ser usadas para descobrir repetições de padrões em sequências

Árvore para duas sequências: exemplo



Sequences:

TAC$_1$

ATA$_2$