Recherche de motifs et ressemblance entre séquences

Algorithmes et structures de données

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Olivier Delgrange
Institut d'Informatique

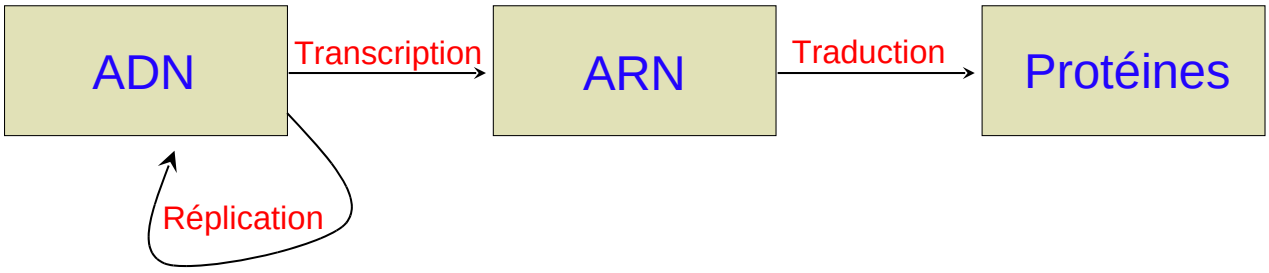
UMONS
Université de Mons

Plan

- Séquences biologiques et chaînes de caractères
- Complexité des algorithmes (rappel)
- Recherche d'un motif dans une séquence
- Arbre des suffixes
- Chaînes répétées et chaînes communes
- Recherche de similarités entre séquences

Séquences biologiques et chaînes de caractères

3 types de séquences biologiques :



Composition :

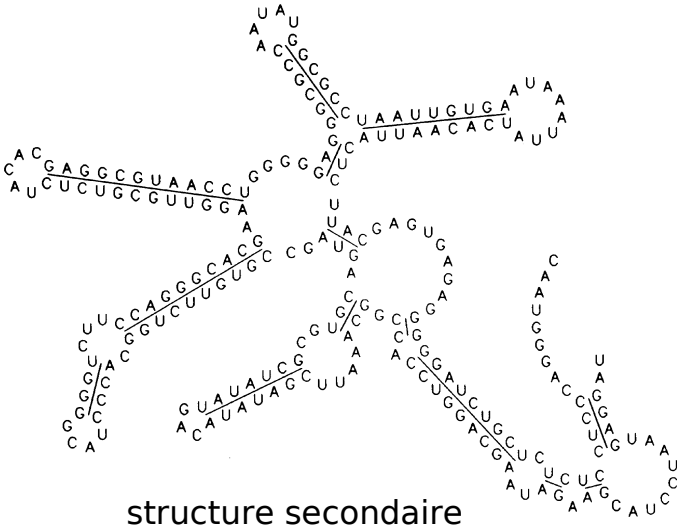
ADN : chaînes de nucléotides (A,C,G,T)
ARN : chaînes de nucléotides (A,C,G,U)
Protéines : chaînes d'acides aminés
(A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y)

Abstraction : on ne considère que les séquences primaires

ADN : Alphabet à 4 lettres : {A,C,G,T}
AATCGGACACTGCGCTTACGGACACCACCTG

ARN : Alphabet à 4 lettres : {A,C,G,U}
CAAUGGGACCCUCCUCUC

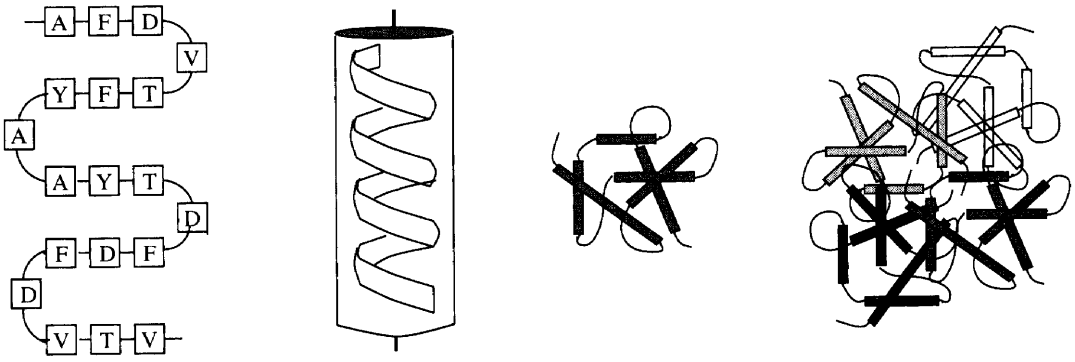
séquences nucléiques



structure secondaire

Protéines : Alphabet à 20 lettres : {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}

AFDVTFYAAYTDFDFDVTVGHCDYAEKLM



Les « séquences génétiques » sont des mots sur un alphabet à 4 lettres (ADN et ARN) ou à 20 lettres (protéines)

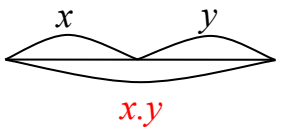
Terminologie

Alphabet A : ensemble fini de lettres (caractères ou symboles)
Mot (séquence) sur A : suite finie de lettres de A
Longueur d'un mot : son nombre de lettres

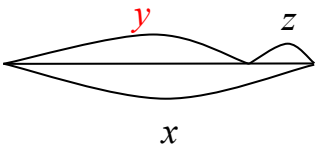
Notation :

Soit A={A,C,G,T} un alphabet, $x \in A^*$ est un mot sur A, $|x|$ est la longueur de x

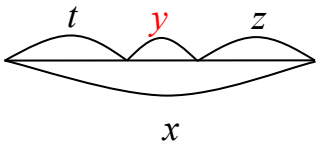
Soit $x,y \in A^*$ avec $x=x_1x_2 \ldots x_n$ et $y=y_1y_2 \ldots y_m$
La concaténation de x et de y est le mot $x.y \in A^*$ avec
$xy= x_1x_2 \ldots x_ny_1y_2 \ldots y_m$



Le mot $y \in A^*$ est préfixe (resp. suffixe) de $x \in A^*$ ssi $\exists\ z \in A^*$: $x=yz$ (resp. $x=zy$). Le mot z peut-être vide ou égal à x.



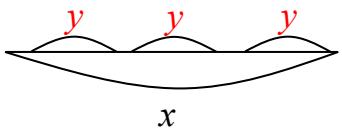
Le mot $y \in A^*$ est facteur de $x \in A^*$ ssi $\exists\ t,z \in A^*$: $x=tyz$.
Les mots t et z peuvent être vides, y peut être vide



Terminologie (2)

Le mot $y \in A^*$ possède une occurrence dans $x \in A^*$ à la position i si $y = x_{i..j}$

Le facteur y de x est répété s'il possède plusieurs occurrences dans x



Le mot $y \in A^*$ est un sous-mot de $x = x_1 x_2 \ldots x_n \in A^*$ si $y = x_{i1} x_{i2} \ldots x_{im}$ avec $m \leq n$ et $1 \leq i_1 < i_2 < \ldots i_m \leq n$

Exemple :

Soit A={A,C,G,T} l'alphabet et x=ACACGTGAGCAGATGCAT.
y=CATGAACT est un sous-mot de x

Que peut-on chercher dans les séquences génétiques ?

Tout ce qui peut nous aider à localiser des zones importantes (telles que des gènes par exemple).

- Des *promoteurs de transcription*
- Des *terminateurs de transcription*
- Des *introns* dans les gènes
- Des *régions d'initiation de traduction*
- Des *motifs (approximativement) répétés*
- Des *motifs statistiquement significatifs*
- Des *zones régulières (complexité de Kolmogorov)*
- Des *motifs répartis tout au long des séquences*
- etc

Complexité d'algorithmes (rappel)

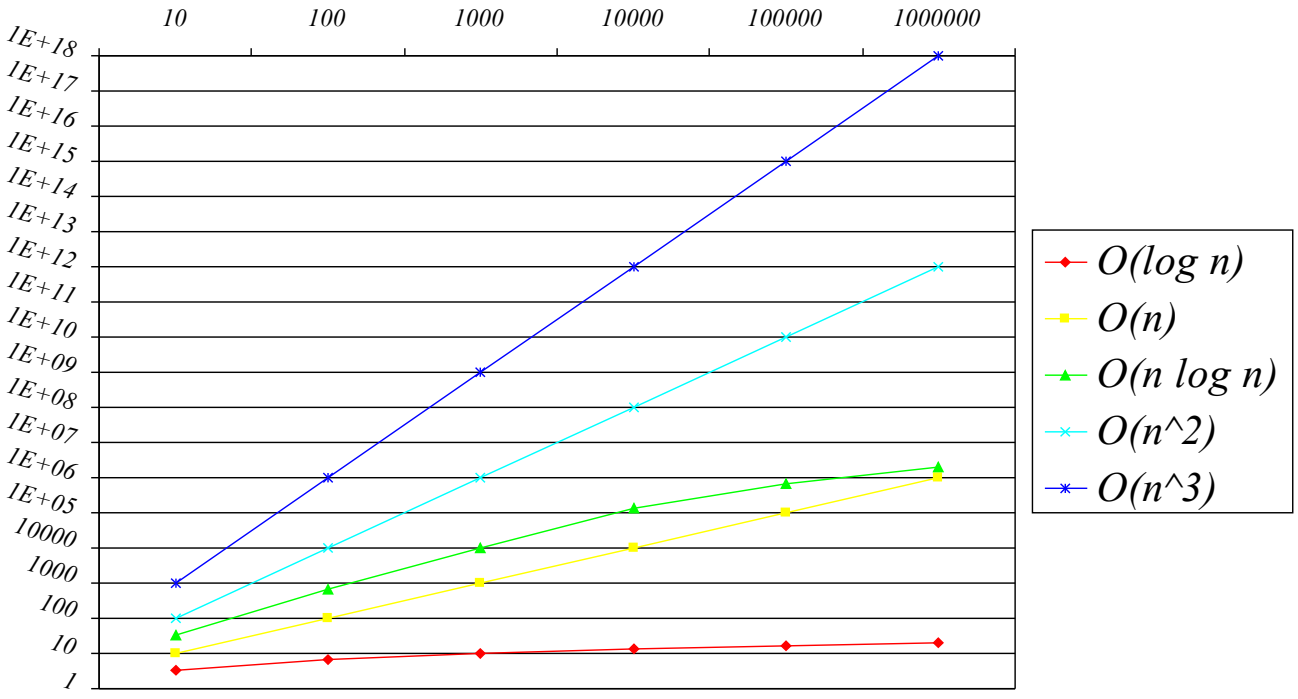
Complexité (en temps) d'un algorithme : nombre d'opérations élémentaires effectuées. Elle s'exprime en fonction de la taille du problème n.

La complexité est $O(f(n))$ si le nombre d'opérations est **borné** par $C.f(n)$, avec C une constante, lorsque n tend vers $+\infty$.

Remarques :

- Seul l'ordre de grandeur du nombre d'opérations nous intéresse.
- On regarde le comportement à l'infini car les données sont souvent de grande taille et on s'intéresse surtout à la croissance de la complexité en fonction de la taille du problème.
- En pratique, la constante C doit être de taille raisonnable. Un algorithme en $O(n^{1.9})$ ne sera pas meilleur qu'un algorithme en $O(n^2)$ si la constante est 10^{15}.
- La complexité en espace évalue de la même manière la quantité de mémoire utilisée par l'algorithme en fonction de la taille du problème.

Complexité d'algorithmes (2)



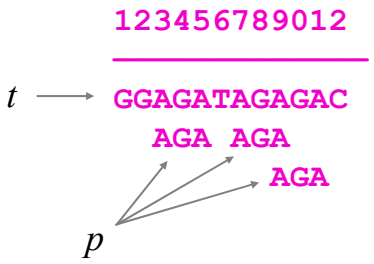
Recherche d'un motif dans une séquence

Position du problème

Soit $p,t \in A^*$ avec $|p| = m$ et $|t| = n$. On veut trouver toutes les occurrences de p (*pattern*) dans t (*texte*). C'est-à-dire trouver toutes les positions h ($1 \leq h \leq n-m+1$) telles que $t_{h..h+m-1} = p$

Exemple :

Soit A={A,C,G,T} l'alphabet et t =GGAGATAGAGAC le texte.
Le motif p =AGA possède 3 occurrences dans t aux positions 3,7 et 9 :



Recherche de signaux dans des séquences génétiques

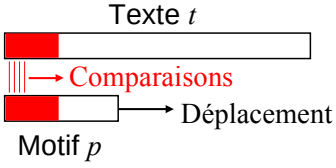
Les séquences peuvent être très longues (≈ 10^9 nucléotides)

↓

Besoin d'algorithmes performants

Méthode naïve

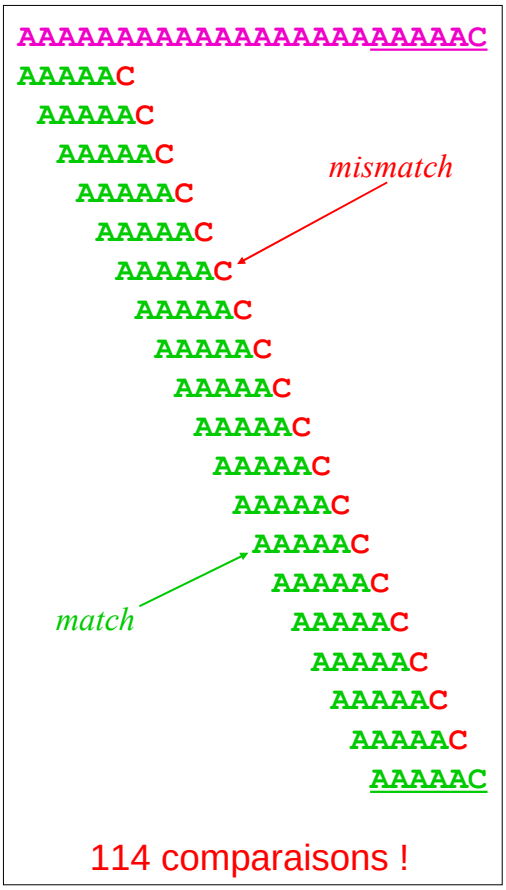
- Le motif p glisse, de la gauche vers la droite, le long du texte t
- Les caractères de p sont comparés avec ceux de t de la gauche vers la droite
- Si différence ⇒décalage de p d'une position vers la droite et reprise des comparaisons au début de p
- Si pas de différence ⇒occurrence trouvée, décalage de p d'une position vers la droite et reprise des comparaisons au début de p
- La recherche est terminée lorsque p a atteint la fin de t



L'algorithme :

```
pour i←1 à n-m+1
    j←1
    tant que j≤ m cet p_j=t_{i+j-1} faire
        j←j+1
    fin tant que
    si j>m alors
        /* occurrence trouvée en position i */
    fin si
fin pour
```

Exécution :



114 comparaisons !

Efficacité :

- On calcule le nombre de comparaisons
- Complexité en temps : $O((n-m+1)*m)=O(nm)$
- L'algorithme est lent car après chaque décalage, il oublie toutes les comparaisons déjà effectuées.

Méthode de Knuth, Morris et Pratt (1977)

- Même principe que l'algorithme naïf

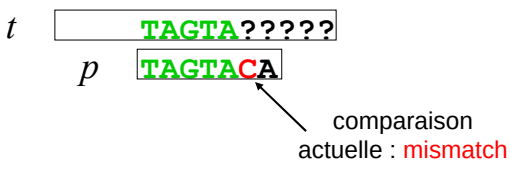
Mais :

- Le décalage peut parfois être de plus d'une position
- Les comparaisons réussies (*matches*) ne sont pas recommencées

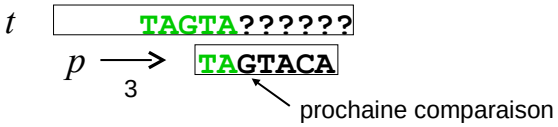
En cas de mismatch, seule la structure du motif détermine la valeur du décalage à appliquer

Exemple : p=TAGTACA

Dans la configuration actuelle :



Quel que soit le texte, on va décaler de 3 positions car le préfixe TA concorde. On reprend les comparaisons après ce TA.



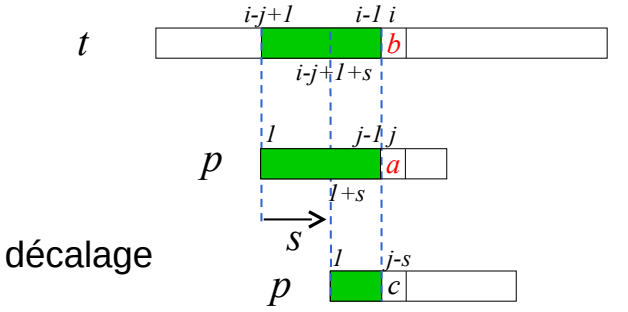
Supposons, dans la recherche, un mismatch entre p_j et t_i : $p_j=a$, $t_i=b$ et $a \neq b$.
Cela signifie que $p_{1..j-1}=t_{i-j+1..i-1}$ et $p_j \neq t_i$.

Le décalage s à appliquer au motif doit être tel que :

A. $p_{1..j-s-1}=t_{i-j+1+s..i-1}$: *les caractères déjà connus de t concordent*

B. $p_{j-s}=c \neq a=p_j$: *un caractère différent de a est amené sous t_i*

C. s est minimal parmi toutes les valeurs qui vérifient A et B
: *aucune occurrence potentielle n'est manquée*



Après le décalage, les comparaisons reprennent entre p_{j-s} et t_i

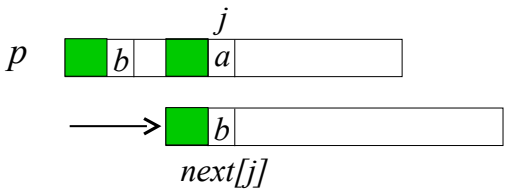
Remarque :

La condition A. s'écrit aussi $p_{1..j-s-1}=p_{1+s..j-1}$

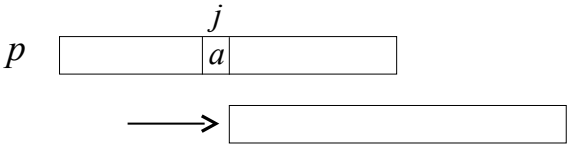
⇒Le décalage lors d'un mismatch est indépendant du texte

Table *next* :

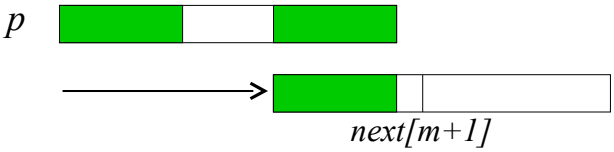
- *next[j]* est défini pour $1 \leq j \leq m+1$
- en cas de mismatch sur le caractère p_j, *next[j]* indique la position dans le motif du caractère qui sera comparé avec le caractère courant du texte après le décalage vérifiant les conditions A, B et C
- une valeur *next[j]=0* indique qu'aucun décalage ne vérifie les conditions et dès lors que le motif doit être décalé au delà de la position courante
- *next[m+1]* est la valeur du décalage à effectuer après avoir trouvé une occurrence. La valeur garantit de ne pas manquer l'occurrence suivante.



Cas 1 : $j<m+1$ et $next[j] \neq 0$



Cas 2 : $j<m+1$ et $next[j]=0$



Cas 3 : $j=m+1$ et $next[j] \neq 0$

p :	TACTGTACTA
next :	01102011053

Exemple

L'algorithme :

```
i←1
j←1
tant que i≤ n faire
    tant que j≠ 0 cet p_j ≠ t_i faire
        j←next[j]
    fin tant que
    i←i+1
    j←j+1
    si j>m alors
        /* occurrence trouvée en position i-j+1 */
        j←next[m+1]
    fin si
fin tant que
```

Exécution :

```
AAAAAAAAAAAAAAAAAAAAAAC
AAAAAC
 AAAAAC
  AAAAAC
   AAAAAC
    AAAAAC
     AAAAAC
      AAAAAC
       AAAAAC
        AAAAAC
         AAAAAC
          AAAAAC
           AAAAAC
            AAAAAC
             AAAAAC
              AAAAAC
               AAAAAC
                AAAAAC
                 AAAAAC
                  AAAAAC
                   AAAAAC
p :      AAAAAC      AAAAAC
next :   0000051     AAAAAC
```

42 comparaisons !

Efficacité :

La complexité en temps est $O(2n)=O(n)$

Preuve :

- Soit nd le nombre de décalages effectués et ni le nombre d'incrémentations de i.
- Le nombre d'étapes est $nd+ni$.
- Puisque $ni\leq n$ et $nd<ni$, le nombre d'étapes est $<2n$.

Problème : le calcul de la table *next* (« preprocessing »)
Solution : adaptation de l'algorithme de Knuth-Morris-Pratt où le motif p glisse le long de lui même.

L'algorithme de preprocessing :

```
i←1
next[1] ←0
j←0
tant que i≤ m faire
    tant que j≠ 0 cet p_i≠ p_j faire
        j←next[j]
    fin tant que
    i←i+1
    j←j+1
    si i>m cou p_i≠ p_j alors
        next[i]←j
    sinon
        next[i]←next[j]
    fin si
fin tant que
```

De la même manière, on montre que la complexité en temps du preprocessing est $O(2m)=O(m)$.

La complexité en temps globale est donc $O(n+m)$

En moyenne, chaque caractère du texte est comparé $1+1/\#A$ fois avec un caractère du motif où $\#A$ est la taille de l'alphabet.

Ce comportement en moyenne est pareil à celui de l'algorithme naïf.

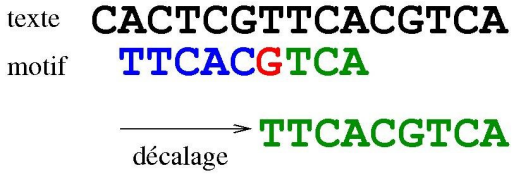
Méthode de Boyer et Moore (1977)

- Le motif p glisse le long du texte t de gauche à droite

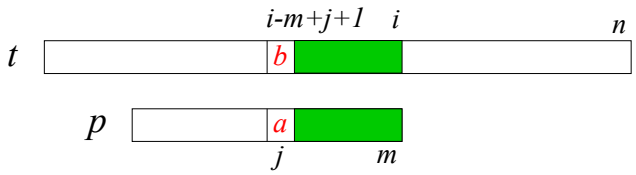
Mais :

- Les caractères sont comparés de droite à gauche !
- Le décalage peut parfois être de plus d'une position

En cas de mismatch, seule la structure du motif détermine la valeur du décalage à appliquer



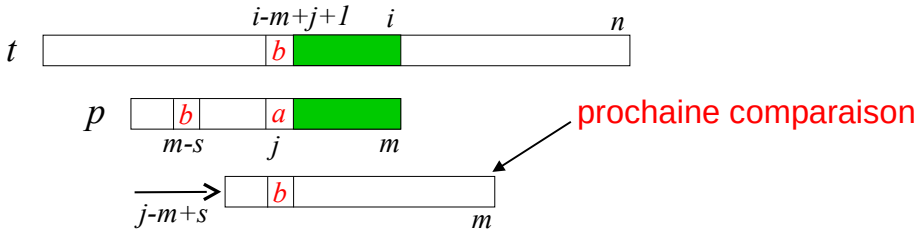
Supposons, dans la recherche que p_m soit aligné avec t_i et qu'un mismatch se produise entre p_j et t_{i-m+j}.
Cela signifie que $p_{j+1..m}=t_{i-m+j+1..i}$ et $p_j \neq t_{i-m+j}$.



Le décalage est obtenu en choisissant le maximum entre les décalages de deux règles heuristiques.
Ensuite, les comparaisons sont reprises en p_m.

Première règle : « Heuristique d'occurrence »

Supposons que p_{m-s} soit l'occurrence la plus à droite du symbole t_{i-m+j} dans p.
Le décalage amène p_{m-s} sous t_{i-m+j} : on décale de $j-m+s$.



La table *skip* nous permet de connaître, pour chaque symbole w de l'alphabet A, la position d'occurrence $m-s$ la plus à droite de w dans p : $skip[w]=m-s$.
Si w n'apparaît pas dans p, alors $skip[w]=m$: p_1 doit être aligné avec $t_{i-m+j+1}$.
La construction de la table *skip* est aisée :

```
pour w←1 à #A
    skip[w]←m
fin pour
pour j←1 à m
    skip[p_j]←m-j
fin pour
```

Temps de calcul : $O(m+\#A)$

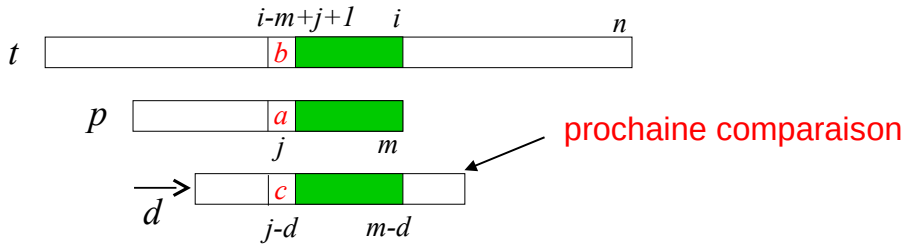
Remarque :

Si $m-s>j$, l'heuristique est inutile !
Le mieux que l'on puisse faire est décaler le motif d'une position vers la droite.

Deuxième règle : « Heuristique de match »

Décalage similaire à celui de Knuth-Morris-Pratt.
Le décalage d à appliquer au motif doit être tel que :

A. $p_{j+1-d..m-d}=t_{i-m+j+1..i}$: *les caractères connus de t concordent*
B. $p_{j-d}=c\neq a=p_j$: *un caractère différent de a est amené sous t_{i-m+j}*
C. d est minimal parmi toutes les valeurs qui vérifient A et B
: *aucune occurrence potentielle n'est manquée*



Remarque :

La condition A. s'écrit aussi $p_{j+1-d..m-d}=p_{j+1..m}$

⇒Le décalage lors d'un mismatch est indépendant du texte

Table *shift* : pour un mismatch en position j de p, $shift[j]$ nous donne le décalage d augmenté de la valeur $(m-j)$ permettant, après décalage, de repositionner le symbole courant de t en face de p_m.

Formellement :

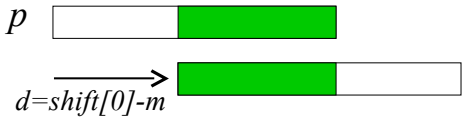
Pour chaque j : $0 \le j \le m$: $shift[j] = min \{ d+m-j \mid d \ge 1$

et ($d \ge j$ ou $p_{j-d} \neq p_j$)

et (pour k : $j < k \le m$: $k-d \le 0$ ou $p_{k-d} = p_k$) }

Remarque :

shift[0] correspond au décalage à effectuer lorsqu'une occurrence a été trouvée. Dans ce cas, le décalage est tel que le plus long préfixe de p (et non égal à p), qui est également suffixe de p, soit aligné avec ce suffixe.



Exemples : soit A={A,C,G,T} l'alphabet

p		A	C	G	T	A	C	G
j	0	1	2	3	4	5	6	7
$m-j$	7	6	5	4	3	2	1	0
d	4	4	4	4	4	7	7	1
$shift[j]$	11	10	9	8	7	9	8	1

$skip[A]=2$ $skip[C]=1$
$skip[G]=0$ $skip[T]=3$

p		A	A	A	A	A	C
j	0	1	2	3	4	5	6
$m-j$	6	5	4	3	2	1	0
d	6	6	6	6	6	6	1
$shift[j]$	12	11	10	9	8	7	1

$skip[A]=1$ $skip[C]=0$
$skip[G]=6$ $skip[T]=6$

L'algorithme :

```
i←m
j←m
tant que i≤ n faire
   si j≠ 0  cet t_i=p_j alors
      i←i-1
      j←j-1
   sinon
      si j=0 alors
            /* Occurrence trouvée en position i+1 */
         i←i+shift[0]
      sinon     /* mismatch */
         i←i+max(skip[t_i],shift[j])
      fin si
      j←m
   fin si
fin tant que
```

Exécution :

```
AAAAAAAAAAAAAAAAAAAAAAC
AAAAAC
 AAAAAC
  AAAAAC
   AAAAAC
    AAAAAC
     AAAAAC
      AAAAAC
       AAAAAC
        AAAAAC
         AAAAAC
          AAAAAC
           AAAAAC
            AAAAAC
             AAAAAC
              AAAAAC
               AAAAAC
                AAAAAC
                 AAAAAC
                  AAAAAC
                   AAAAAC
                    AAAAAC
```

24 comparaisons !

Efficacité : (sans preuve !)

Lorsque p n'apparaît pas dans t, la complexité en temps est $O(3n)=O(n)$.

Si p apparaît dans t, la complexité en temps est $O(n+rm)$, où r est le nombre d'occurrences de p dans t $\Rightarrow$dans le pire des cas $O(nm)$.

Mais :

En moyenne, chaque caractère du texte est comparé $1/\#A$ fois avec un caractère du motif $\Rightarrow$algorithme sous-linéaire en pratique.

L'algorithme de preprocessing :

(pour information)

Problème : le calcul de la table *shift* (« preprocessing »)

La complexité en temps du preprocessing est $O(m)$

Il existe beaucoup de variantes très alambiquées de l'algorithme de Boyer-Moore, ou de son preprocessing, visant à en améliorer la complexité.

```
pour i←1 à m
   shift[i]←2m-i
fin pour
j←m ; k←m+1
tant que j≥ 0 faire
   f[j]←k
   tant que k≤ m cet p_j≠ p_k faire
      shift[k]←min{shift[k],m-j}
      k←f[k]
   fin tant que
   j←j-1  ;  k←k-1
fin tant que
pour i←0 à k
   shift[i]←min{shift[i],m+k-i}
fin pour
j←f[k]
tant que k≤ m faire
   tant que k≤ j faire
      shift[k]←min{shift[k],j-k+m}
      k←k+1
   fin tant que
   j←f[j]
fin tant que
```

Arbre des suffixes

Soit $x \in A^*$, avec $|x| = n$.
L'arbre des suffixes du « texte » x, noté $ST(x)$, est un index compact de tous les facteurs de x.

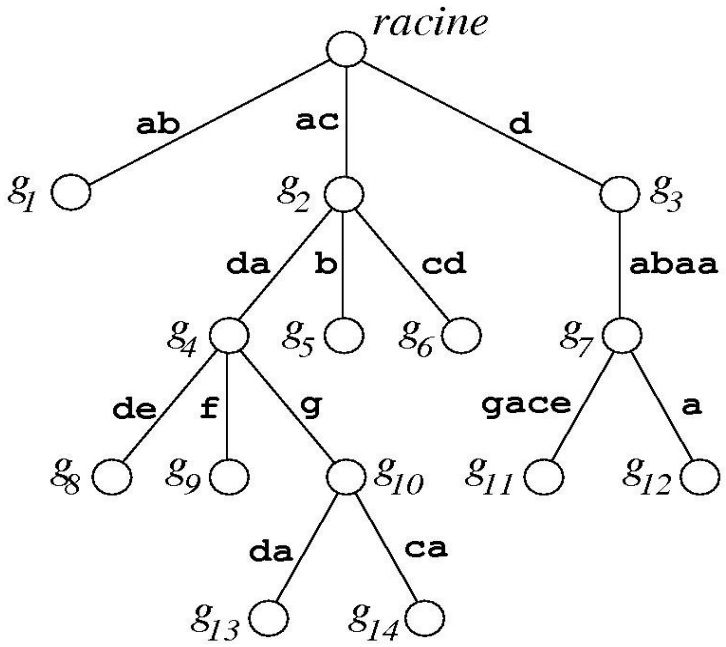
Rechercher la présence de $p \in A^*$, avec $|p| = m$, dans x se fait en temps $O(m)$ à l'aide de $ST(x)$.

Si on doit rechercher la présence de plusieurs motifs dans un même texte x, on construit $ST(x)$ une seule fois et ensuite le temps de recherche est proportionnel à la somme des longueurs des motifs

Espace mémoire pour le stockage de $ST(x)$: $O(n)$.
Algorithmes de construction de $ST(x)$: temps $O(n)$.
(*Weiner, McCreight, Ukkonen*)

Qu'est-ce que l'arbre des suffixes $ST(x)$?

Rappels sur les arbres



Concepts :

nœud
arc : *arc(p,f)*
arc entrant et sortant
étiquette : *etiq(arc(p,f))*
père : *pere(f)*
fils : *fils(p)*
racine
feuille
nœud interne
descendant
ancêtre
degré : *degre(p)*
chemin : *chemin(p,q)*
étiquette de chemin : *etiq(chemin(p,q))*

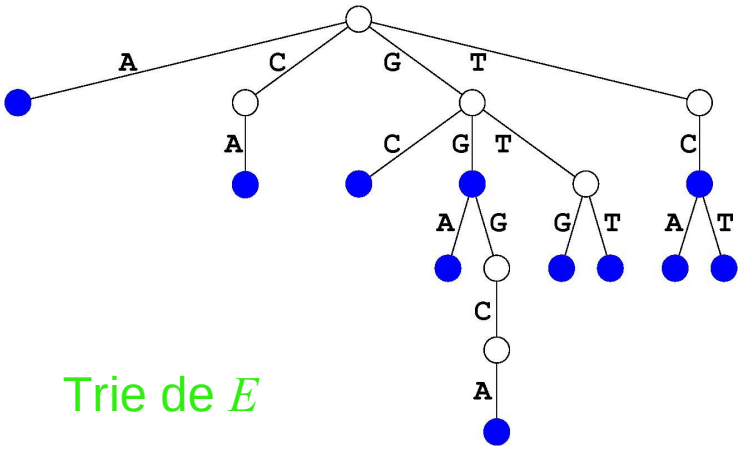
Trie de l'ensemble de mots $E \subset A^*$: arbre qui vérifie les propriétés :

1. pour tous les arcs $arc(p,f)$: $etiq(arc(p,f)) \in A$
2. pour tout nœud p, si $f_1 f_2 \in fils(p)$ et $etiq(arc(p,f_1)) = etiq(arc(p,f_2))$ alors $f_1 = f_2$
3. il existe deux types de nœud : les nœuds terminaux et les nœuds non terminaux. Toutes les feuilles sont des nœuds terminaux
4. si g nœud terminal alors il existe $w \in E$: $etiq(chemin(racine,g)) = w$
5. si $w \in E$ alors il existe g nœud terminal : $etiq(chemin(racine,g)) = w$
6. si f nœud non terminal alors il existe $w \in E$: $etiq(chemin(racine,f))$ préfixe de w

Pour $y \in A^*$, déterminer si y est dans E se fait en temps $O(|y|)$

Exemple :

Soit A={A,C,G,T} l'alphabet
Soit E={A,CA,GC,GG,GGA, GGGCA,GTG,GTT,TC, TCA,TCT}



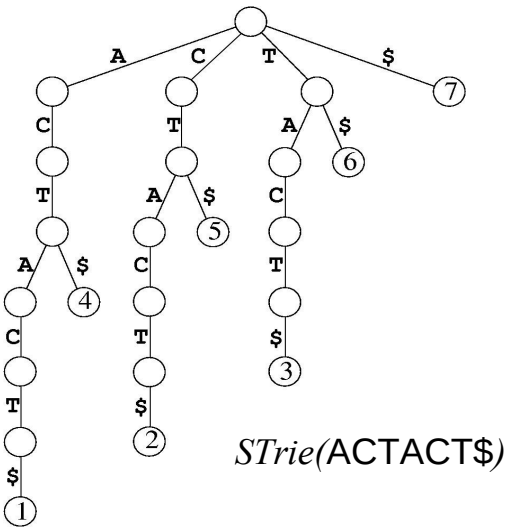
Trie de E

Trie des suffixes de $x \in A^*$, avec $|x| = n > 0$

Supposons que le dernier symbole de x soit \$, différent de tous les autres.
NB : Si ce n'est pas le cas, il suffit d'étendre l'alphabet : utiliser $A \cup \{\$\}$.
Soit S_x l'ensemble de tous les suffixes de x.
Le trie des suffixes de x, noté $STrie(x)$ est le trie de S_x.

Exemple :

x=ACTACT\$ $\Rightarrow S_x$={ACTACT\$,CTACT\$,TACT\$,ACT\$,CT\$,T\$,\$}



$STrie$(ACTACT\$)

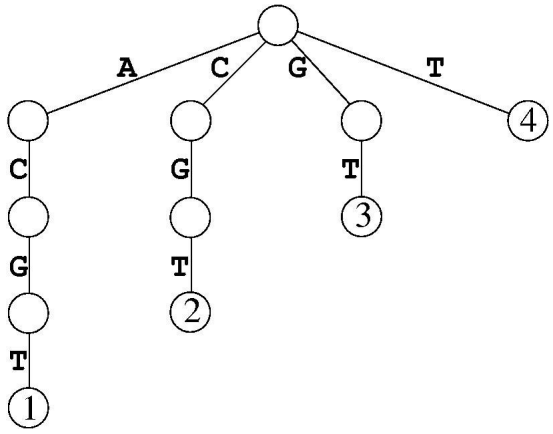
- tous les nœuds terminaux sont des feuilles
- chaque feuille correspond à un suffixe. On les numérote par la position de début dans x
- chaque nœud interne correspond à un facteur
- concepts : $fact(g)$ et $locus(w)$

- Pour $y \in A^*$, déterminer si y est facteur de x se fait en temps $O(|y|)$

Désavantage du trie des suffixes :

coût mémoire : $\frac{n^2+n}{2}$ nœuds dans le pire des cas

*STrie(*ACGT*) :*

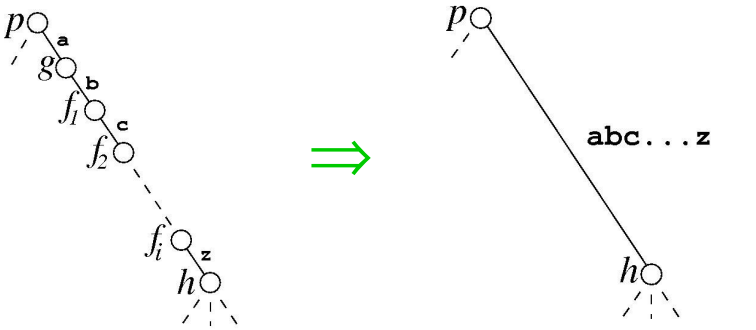


Arbre des suffixes : *ST(x)*

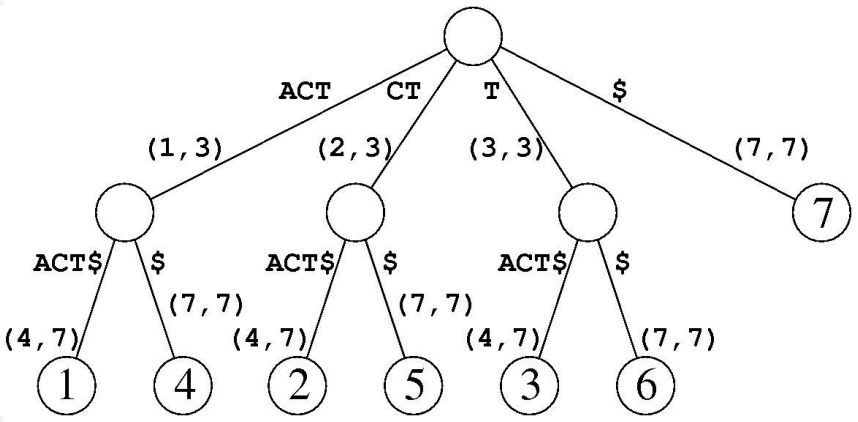
ST(x) est une version compacte de *STrie(x)* :
seuls les « nœuds importants » sont conservés :

- les feuilles : si h est une feuille de *STrie(x)* alors *fact(h)* est suffixe de x
- les nœuds de degré ≥ 2 : supposons g un nœud de *STrie(x)* avec *degre(g)*≥ 2
  Soit $l=|fact(g)|$ et $f_1,f_2 \in fils(g)$ avec $etiq(arc(g,f_1)) \neq etiq(arc(g,f_2))$
  ⇒il existe 2 suffixes $x_{i..n}$ et $x_{j..n}$ tels que :
  - *fact(g)* est le préfixe de longueur l de $x_{i..n}$ **et de** $x_{j..n}$
  - *fact(f_1)* est le préfixe de longueur $l+1$ de $x_{i..n}$ **mais pas de** $x_{j..n}$
  - *fact(f_2)* est le préfixe de longueur $l+1$ de $x_{j..n}$ **mais pas de** $x_{i..n}$

On supprime tous les nœuds de degré 1 en concaténant les étiquettes :



- les étiquettes pour un même père commencent par des caractères différents



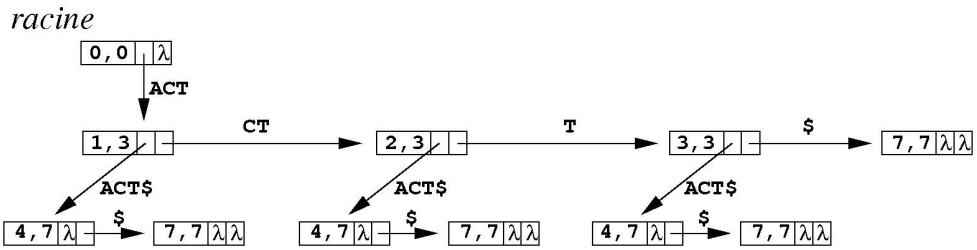
*ST(*ACTACT$*)*

- étiquette : *(début,fin)* du facteur
- nombre maximal de nœuds : *2n-1*
- concepts : *locus étendu* et *locus contracté*
- chaque nœud interne est la « séparation » entre deux suffixes : « répétition maximale à droite »

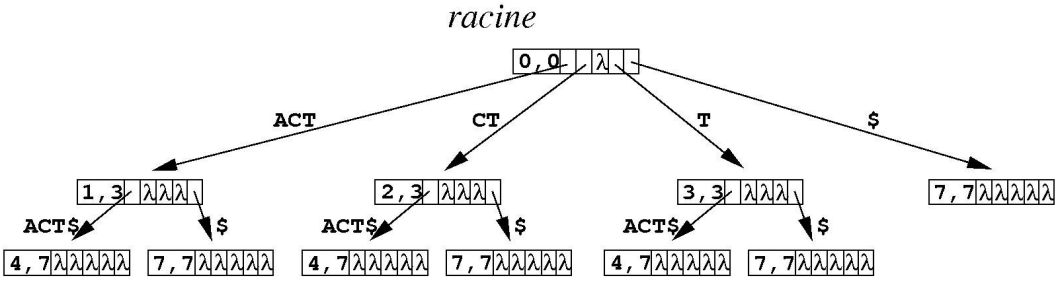
Implémentation de l'arbre des suffixes

Pour chaque arc, l'étiquette est mémorisée dans le nœud fils

1er cas : l'alphabet est grand ou de taille inconnue ⇒temps d'accès : $O(\#A)$



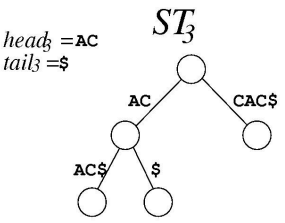
2ème cas : l'alphabet est petit et connu ⇒temps d'accès : $O(1)$

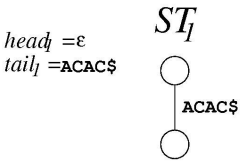


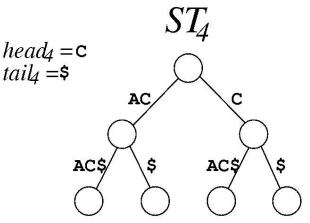
Construction de l'arbre des suffixes : algorithme en force brute

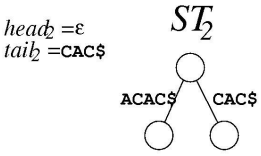
- Les suffixes sont insérés dans l'arbre en allant du plus long au plus court
- l'arbre de départ ST_0 ne contient que la racine
- à l'étape i, le suffixe $x_{i..n}$ est scindé en $x_{i..n}=head_i.tail_i$ où $head_i$ est le plus long préfixe de $x_{i..n}$ également préfixe d'un suffixe $x_{j..n}$ présent dans ST_{i-1} (donc $j<i$)
- l'étape i crée $locus(head_i)$ s'il n'existait pas et $locus(x_{i..n})$ dans tous les cas
- l'algorithme fonctionne en temps $O(n+(n-1)+(n-2)+...1)=O(n^2)$

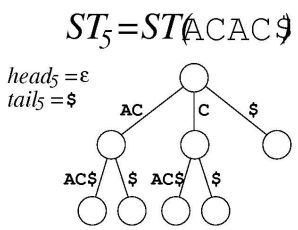
Exemple : x=AAAAAAAA$











Algorithme de McCreight (1976)

À l'étape i, l'algorithme en force brute recherche $head_i$ en parcourant l'arbre à partir de la racine.

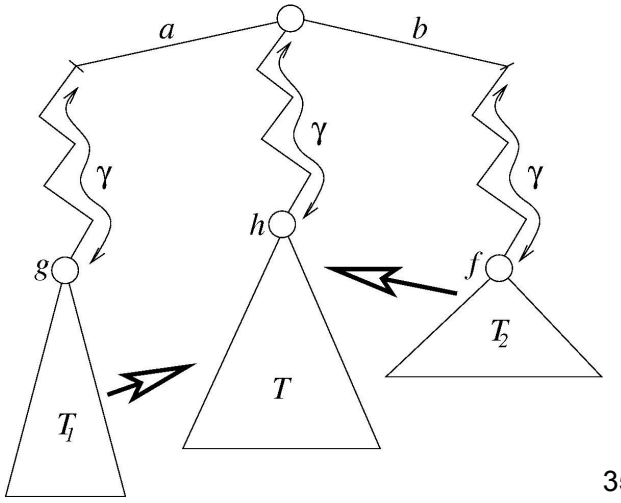
Pour gagner du temps, l'algorithme de McCreight utilise une propriété qui lie le suffixe $x_{i..n}$ au suffixe précédent $x_{i-1..n}$.

Propriété 1 :
Si g, locus de $a.\gamma$, avec $a \in A$ et $\gamma \in A*$ est un nœud interne de $ST(x)$, alors le nœud h, locus de γ , existe et est également un nœud interne de $ST(x)$.

NB : les descendants de g et de h ne sont cependant pas pareils.

Dans $ST(x)$, on associe à chaque nœud interne $g=locus(a\gamma)$ un pointeur vers le nœud interne $h=locus(\gamma)$: le lien suffixe.

Notation : $h=liensuf(g)$

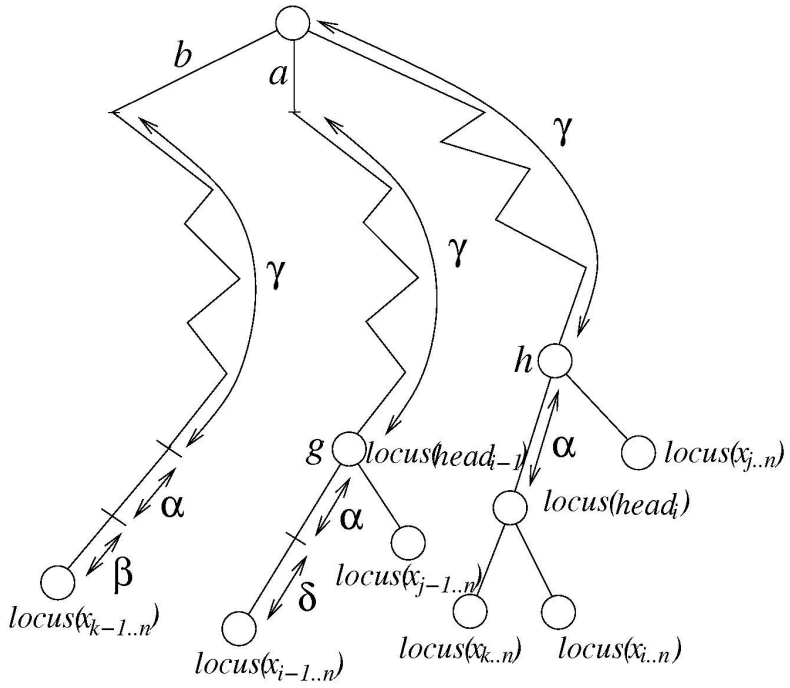


Malheureusement, les liens suffixes ne sont pas tous connus pour les arbres intermédiaires ST_i.

Propriété 2 :
À la fin de l'étape i, après insertion de $x_{i..n}$ dans l'arbre des suffixes intermédiaire, si $head_{i-1}$ est non vide alors la valeur à donner à $liensuf(locus(head_{i-1}))$ est connue.

Preuve : Posons $head_{i-1}=a\gamma$, avec $a \in A$ et $\gamma \in A*$. Le mot $a\gamma$ est donc le plus long préfixe commun entre $x_{i-1..n}$ et $x_{j-1..n}$, avec $j-1<i-1$. Donc γ est le plus long préfixe commun entre $x_{i..n}$ et $x_{j..n}$.
- si $\gamma = head_i$, alors $locus(\gamma)$ est créé à l'étape i.
- si $\gamma \neq head_i$, alors γ est préfixe de $head_i$ et $head_i$ est le plus long préfixe commun entre $x_{i..n}$ et $x_{k..n}$, avec $k<i$. Dès lors, γ est le plus long préfixe commun entre $x_{j..n}$ et $x_{i..n}$. Le nœud $locus(\gamma)$ existe donc, il a été créé à une étape antérieure.

La valeur à donner à $liensuf(locus(head_{i-1}))$ à la fin de l'étape i est donc $locus(\gamma)$.



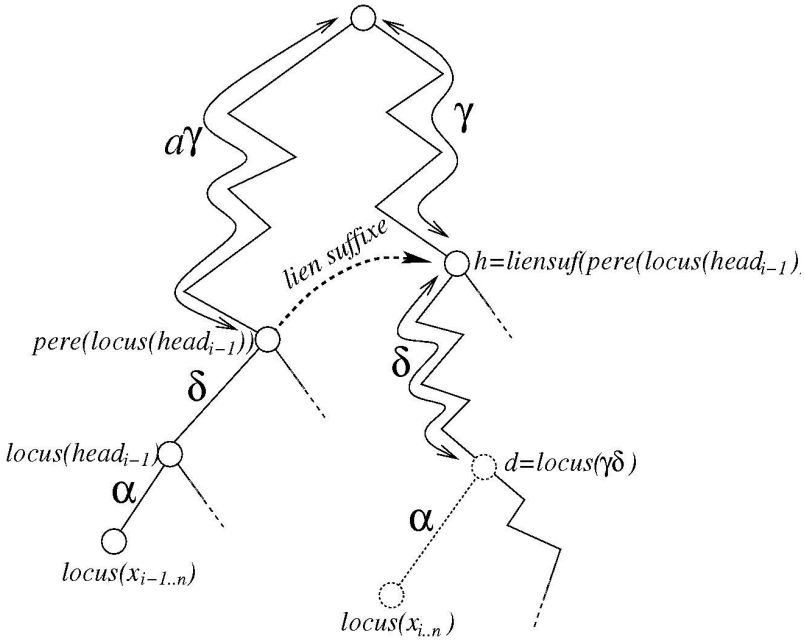
À l'étape i, pour trouver $head_i$, on peut donc se servir des valeurs des $liensuf(g)$, avec g nœud interne autre que la racine et autre que $locus(head_{i-1})$

Etape i :

1er cas : ni $locus(head_{i-1})$ ni $pere(locus(head_{i-1}))$ n'est la racine.

Soit $head_{i-1}=a\gamma\ \delta$, avec $a \in A$, γ ,$\delta \in A^*$ et $\delta = etiq(arc(pere(locus(head_{i-1})),locus(head_{i-1})))$

Première amélioration : rechercher $head_i$ à partir de $h=liensuf(pere(locus(head_{i-1})))$



Le mot $\gamma\ \delta$ est préfixe de $x_{i..n}$.

Le mot $\gamma\ \delta$ est préfixe d'un suffixe déjà représenté dans l'arbre actuel.

Donc il est rapide de trouver le locus contracté de $\gamma\ \delta$ à partir du nœud h : il suffit de tester une lettre par arc.

Si le locus de $\gamma\ \delta$ n'existe pas dans l'arbre, alors $head_i = \gamma\ \delta$.
On peut donc créer $locus(head_i)$, $locus(x_{i..n})$ et donner la valeur $locus(head_i)$ à $liensuf(locus(head_{i-1}))$.

Si le locus de $\gamma\ \delta$ existe dans l'arbre alors $\gamma\ \delta$ n'est pas forcément le plus long préfixe entre un suffixe déjà présent et $x_{i..n}$.
Soit α le mot tel que $x_{i-1..n} = a\gamma\ \delta\ \alpha$ et $x_{i..n} = \gamma\ \delta\ \alpha$.
Il faut alors rechercher $head_i$, à partir du nœud $d = locus(\gamma\ \delta)$, en parcourant les caractères de α un à un comme dans le cas de l'algorithme en force brute.
On crée alors $locus(head_i)$, $locus(x_{i..n})$ et on donne la valeur $locus(\gamma\ \delta)$ à $liensuf(locus(head_{i-1}))$.

2ème cas : $pere(locus(head_{i-1}))$ est la racine.

Soit $head_{i-1} = a\delta$ et $x_{i..n} = \delta\ \alpha$ avec $a \in A$, δ ,$\alpha \in A^*$.
En suivant le même raisonnement, δ est préfixe d'un mot déjà représenté dans l'arbre et si son locus n'existe pas alors $head_i = \delta$.
On peut donc créer $locus(head_i)$, $locus(x_{i..n})$ et donner la valeur $locus(head_i)$ à $liensuf(locus(head_{i-1}))$.

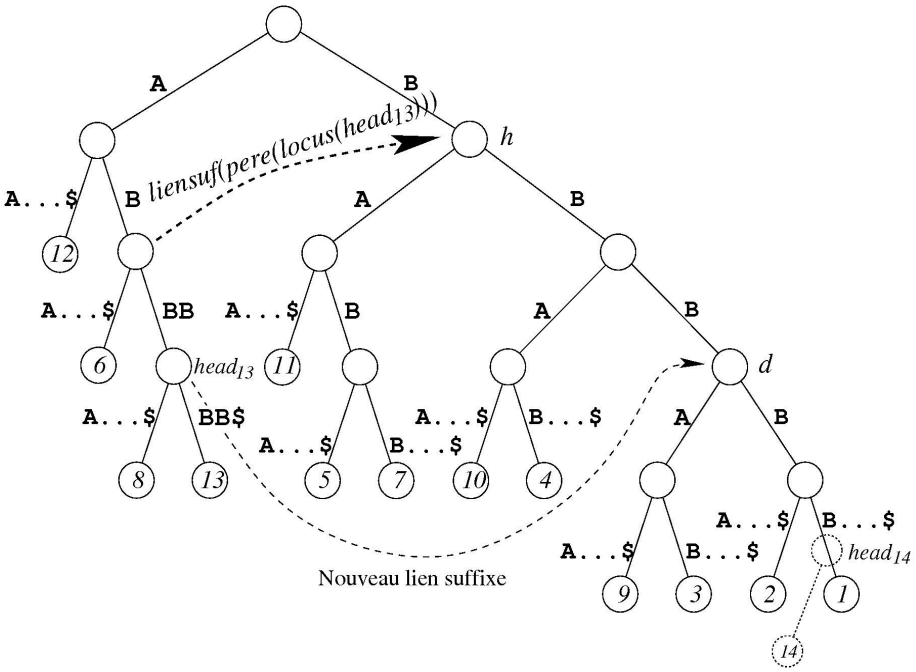
Si le locus de δ existe dans l'arbre alors δ n'est pas forcément le plus long préfixe entre un suffixe déjà présent et $x_{i..n}$.
Il faut alors rechercher $head_i$, à partir du nœud $d = locus(\delta)$, en parcourant les caractères de α un à un comme dans le cas de l'algorithme en force brute.
On crée alors $locus(head_i)$, $locus(x_{i..n})$ et on donne la valeur $locus(\delta)$ à $liensuf(locus(head_{i-1}))$.

3ème cas : $locus(head_{i-1})$ est la racine.

Il n'est alors pas possible de faire mieux que l'algorithme en force brute : on recherche $head_i$ à partir de la racine en parcourant les caractères de $x_{i..n}$ un à un.

Dans ce cas, il ne faut pas créer de lien suffixe car la racine n'en possède pas.

Exemple : x=BBBBBABABBBAABBBBB$ sur A={A,B,$}



Arbre intermédiaire lors de la 14ème étape.

Efficacité :

L'algorithme de McCreight construit l'arbre des suffixes de $x=x_1x_2...x_n$, avec $x_n \neq x_i$, $\forall i$: $1 \leq i < n$, en temps $O(n)$

Autres algorithmes :

- l'algorithme de Weiner (1973) construit l'arbre des suffixes en insérant les suffixes du plus court au plus long.
  Temps de construction : $O(n)$
  L'algorithme de Weiner est plus coûteux en espace mémoire que l'algorithme de McCreight : $O(\#A.n)$

- l'algorithme de Ukkonen (1992) construit l'arbre des suffixes « on-line », c'est-à-dire que les caractères de x sont parcourus de gauche à droite et que chaque étape fournit un arbre des suffixes valide pour la partie de x déjà traitée.
  Temps de construction : $O(n)$
  Espace mémoire : $O(n)$
  Il s'agit d'un algorithme complexe.

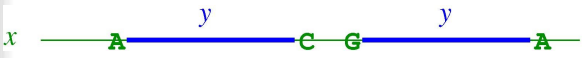
Chaînes répétées et chaînes communes

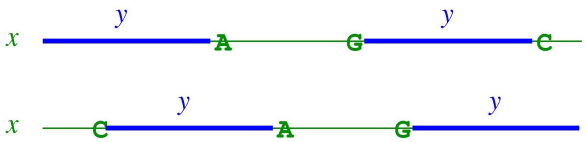
Utilité en bio-informatique : donner des renseignements sur l'évolution d'une séquence ou sur le passé commun entre deux séquences.

Qu'est-ce qu'une répétition maximale ?

Soit $x,y \in A^*$ avec $|x|=n$, $|y|=m$ et y possédant deux occurrences dans x : $y=x_i...x_{i+m-1}$ et $y=x_j...x_{j+m-1}$, avec $i \neq j$. La répétition de y est maximale si les deux occurrences ont un contexte gauche et droit différent, c'est-à-dire si :
contexte gauche différent : $i=0$ ou $j=0$ ou $x_{i-1} \neq x_{j-1}$
contexte droit différent : $i=n-m+1$ ou $j=n-m+1$ ou $x_{i+m} \neq x_{j+m}$

Si un seul des deux contextes est différent, on parle de répétition maximale à gauche (resp. à droite).





Premier problème :

Soit $x \in A^*$, avec $|x|=n$. Trouver le plus long facteur répété de x

Exemple :
Si x=GCTTAGGTCAGCTACAGGTCAACG, la plus longue répétition concerne le facteur AGGTCA (positions 5 et 16).

Remarque : la plus longue répétition est toujours maximale !

Méthode en force brute

Imaginons la matrice $n \times n$ M dont les valeurs sont 1 ou 0 définie par

$$M_{i,j} = \begin{cases} 0 \text{ si } x_i = x_j \\ 1 \text{ si } x_i \neq x_j \end{cases}$$

Remarque : il s'agit d'une matrice symétrique, on ne s'intéressera qu'à la moitié au dessus de la diagonale.

```
                          1                   2
  | 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
  | G C T T A G G T C A G C T A C A G G T C A A C G
--|-------------------------------------------------
G | 1         1 1       1           1 1           1
C |   1             1     1     1         1     1
T |     1 1       1         1           1
T |     1 1       1         1           1
A |         1         1       1   1         1 1
G | 1         1 1       1           1 1           1
G | 1         1 1       1           1 1           1
T |     1 1       1         1           1
C |   1             1     1     1         1     1
A |         1         1       1   1         1 1
G | 1         1 1       1           1 1           1
C |   1             1     1     1         1     1
T |     1 1       1         1           1
A |         1         1       1   1         1 1
C |   1             1     1     1         1     1
A |         1         1       1   1         1 1
G | 1         1 1       1           1 1           1
G | 1         1 1       1           1 1           1
T |     1 1       1         1           1
C |   1             1     1     1         1     1
A |         1         1       1   1         1 1
A |         1         1       1   1         1 1
C |   1             1     1     1         1     1
G | 1         1 1       1           1 1           1
```

La plus longue répétition est déterminée par la plus longue suite de 1, située sur une autre diagonale que la diagonale principale

L'algorithme (on ne construit pas la matrice) :

```
pluslong←0
pour d←2 à n  /* num. diagonale */
   debi←0
   j←1
   pour i←d  à  n
      si (debi≠ 0)  alors
         si (xi=xj) alors
            long←long+1
         sinon
            si (long>pluslong) alors
               debut1←debj
               debut2←debi
               pluslong←long
            fin si
            debi←0
         fin si
      sinon
         si (xi=xj) alors
            long←1
            debi←i
            debj←j
         fin si
      fin si
      j←j+1
   fin pour
   si (long>pluslong) alors
      debut1←debj
      debut2←debi
      pluslong←long
   fin si
fin pour      /* diagonale d */
```

Résultat : *debut1*, *debut2* et *long*

Efficacité :

- complexité en temps : $O(n^2)$
- complexité en espace : $O(n)$

Utilisation de l'arbre des suffixes

Propriété 1 :

Dans l'arbre des suffixes $ST(x\$)$, si g est un nœud interne alors $fact(g)$ est un facteur répété de x.

Preuve : $fact(g)$ est préfixe de plusieurs suffixes de x car $degre(g)>1$.

Il suffit donc de rechercher dans l'arbre des suffixes le nœud interne locus du plus long mot. Cela peut se faire lors de la construction ou par une recherche en temps $O(n)$ après la construction.

Propriété 2 :

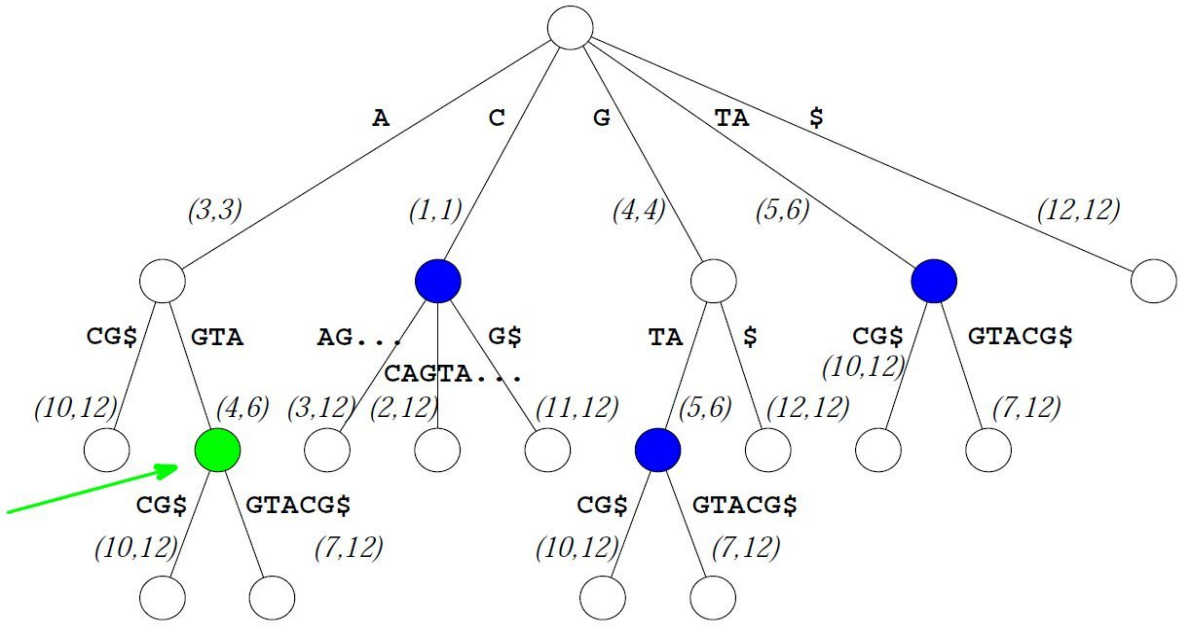
Soit g un nœud interne de $ST(x\$)$. Si il existe $f_l \in fils(g)$ avec $degre(f_l) \neq 0$, alors f_l est locus d'un facteur répété plus long que $fact(g)$.

Preuve : $fact(g)$ est préfixe de $fact(f_l)$. Puisque $degre(f_l) \neq 0$, $fact(f_l)$ est un facteur répété de x.

Il suffit donc de rechercher, parmi tous les nœuds internes, autres que la racine et n'ayant que des feuilles comme fils, celui correspondant au facteur le plus long

Efficacité : Recherche en temps $O(n)$: la recherche est effectuée en un seul parcours récursif de $ST(x\$)$.

Remarque : tout nœud interne est le locus d'une répétition maximale à droite. Puisqu'elle est la plus longue, elle est aussi maximale à gauche.



Deuxième problème :

Soit $x,y \in A^*$, avec $|x|=n$ et $|y|=m$.
Trouver le plus long facteur commun à x et à y

Exemple : x=ACG et y=CACT

Construction de $ST(x\#y\$)$, avec $\#,\$ \notin A$

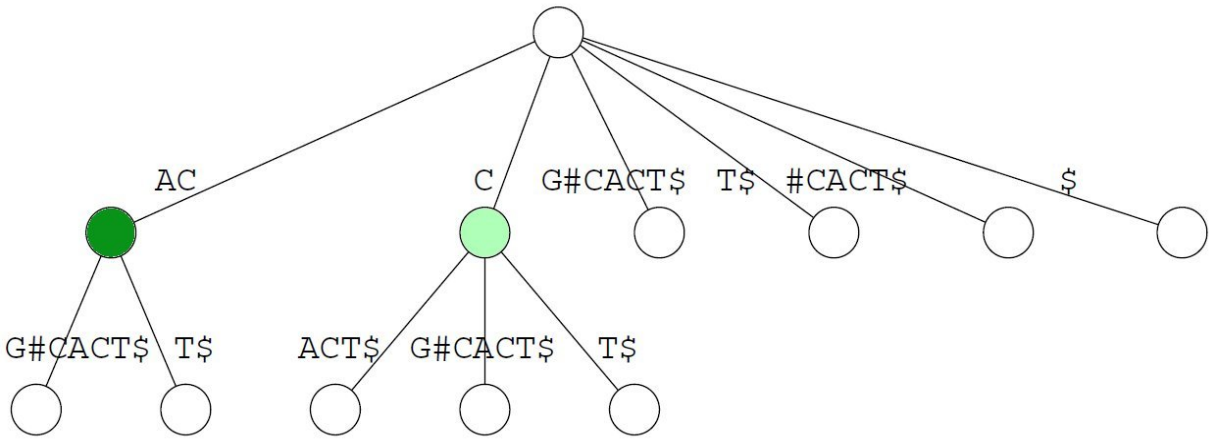
Un nœud g de $ST(x\#y\$)$, est candidat à être locus du plus long facteur commun si :

- g est un nœud interne autre que la racine
- il existe une feuille f_1 dans le sous-arbre dont g est la racine telle que $etiq(chemin(g,f_1))$ **contient** #
- il existe une feuille f_2 dans le sous-arbre dont g est la racine telle que $etiq(chemin(g,f_2))$ **ne contient pas** #.

Lors d'un parcours récursif de $ST(x\#y\$)$, un nœud interne g est candidat si :

- pour au moins un de ses fils f, $etiq(arc(g,f))$ contient # ou f possède parmi ses descendants une feuille f_1 telle que $etiq(chemin(f,f_1))$ contient #
- ET pour au moins un de ses fils f', f' possède parmi ses descendants une feuille f_2 telle que $etiq(chemin(f',f_2))$ ne contient pas # et $etiq(arc(g,f'))$

ne contient p



$ST($ACG#CACT$$)$

Efficacité :

- complexité en temps : $O(n+m)$
- complexité en espace : $O(n+m)$

À chaque nœud, on ajoute les champs *diese* (resp. *sansdiese*) permettant de mémoriser s'il existe un chemin avec # (resp. sans #) dans la descendance.

L'algorithme (informel) :

```
Proc TrouverPlusLong(nœud)
si (nœud est une feuille) alors
   nœud.diese←faux
   nœud.sansdiese←vrai
sinon
   nœud.diese←faux
   nœud.sansdiese←faux
   pour chaque f fils de nœud
      si (etiq(arc(nœud,f)) contient #) alors
         nœud.diese←vrai    /* il ne sert à rien de descendre dans f */
      sinon
         TrouverPlusLong(f)
         si (f.diese) alors nœud.diese←vrai fin si
         si (f.sansdiese) alors nœud.sansdiese←vrai fin si
      fin si
   fin pour
   si (nœud.diese) et (nœud.sansdiese) alors
      /* nœud est candidat pour être le locus du plus long facteur commun */
   fin si
fin si
```

Remarque : si un descendant de nœud est aussi candidat alors nœud ne sera pas le locus du plus long.

Recherche de similarités entre séquences

Base de beaucoup de résolutions de problèmes de bio-informatique (phylogénie, assemblage de fragments, détermination de la fonction d'une séquence...)

Consiste à déterminer les parties qui se ressemblent et les parties qui diffèrent entre plusieurs séquences.

Utilité en biologie

- déduire la fonction d'une séquence en fonction d'autres : deux séquences proches du point de vue de leur suite de symboles ont une fonction proche
- mettre en évidence les séquences ancêtres communes entre plusieurs séquences
- assembler plusieurs fragments en une super-séquence en exploitant les chevauchements approximatifs entre les fragments
- mettre en évidence les différences de séquençage entre les résultats d'un laboratoire et les résultats d'un autre
- etc

Comparaison globale : alignement et alignement optimal

Soit $s,t \in A^*$ avec $|s|=m$ et $|t|=n$, les deux séquences dont on veuille évaluer la ressemblance.

On « aligne » une séquence au dessus de l'autre en insérant des espaces pour les amener à la même longueur.

Alignement :

```
GA-CGGATTAG
GATCGGAATAG
```

En première approximation, on vise à mettre en correspondance un maximum de lettres identiques.

Un espace (−) ne peut pas se trouver aligné en face d'un autre espace, la longueur maximale des chaînes ainsi créées est donc $n+m$.

- une paire alignée $\begin{smallmatrix} - \\ a \end{smallmatrix}$ ou $\begin{smallmatrix} a \\ - \end{smallmatrix}$, avec $a \in A$ est appelée un gap
- une paire alignée $\begin{smallmatrix} a \\ b \end{smallmatrix}$, avec $a,b \in A$ et $a \neq b$ est appelée un mismatch (ou substitution)
- une paire alignée $\begin{smallmatrix} a \\ a \end{smallmatrix}$, avec $a \in A$ est appelée un match

Le score d'un alignement est la somme des scores des paires alignées.

- soit g le score d'un gap
- soit $p(a,b)$ le score d'une paire alignée $\begin{smallmatrix} a \\ b \end{smallmatrix}$ avec $a,b \in A$

Exemple :

Prenons $g=-2$, $p(a,a)=1$ et $p(a,b)=-1$ si $a \neq b$ (souvent utilisé en pratique)

Le coût de l'alignement
```
GA-CGGATTAG
GATCGGAATAG
```
est 6.

Un alignement optimal de s et t est un alignement de s et t de score maximal

La similarité de s et t, notée $sim(s,t)$ est le score de tout alignement optimal entre s et t

Comment calculer la similarité entre deux séquences ?

- **Première méthode** : engendrer tous les alignements possibles et en choisir un de meilleur score ⇒ Impraticable !

• **Deuxième méthode** : programmation dynamique

La recherche de $sim(s,t)$ utilise la connaissance des $sim(s',t')$, avec s' préfixe de s et t' préfixe de t.

Il est facile de voir que $sim(s,t)=max\begin{cases} sim(s,t_{1..n-1})+g \\ sim(s_{1..m-1},t_{1..n-1})+p(s_m,t_n) \\ sim(s_{1..m-1},t)+g \end{cases}$

D'une manière générale : $sim(s_{1..i},t_{1..j})=max\begin{cases} sim(s_{1..i},t_{1..j-1})+g \\ sim(s_{1..i-1},t_{1..j-1})+p(s_i,t_j) \\ sim(s_{1..i-1},t_{1..j})+g \end{cases}$, avec $2\le i\le m$ et $2\le j\le n$

Soit a la matrice $(m+1)\times(n+1)$ telle que $a_{i,j}=sim(s_{1..i},t_{1..j})$ pour $1\le i\le m$ et $1\le j\le n$.

Définissons $a_{i,0}=i\times g$ et $a_{0,j}=j\times g$ pour $1\le i\le m$ *et* $1\le j\le n$.
Ces valeurs initiales correspondent aux alignements débutant par une succession de gaps.
Elles permettent de calculer $a_{1,j}$ et $a_{i,1}$ par la même formule de récurrence.

Le calcul de $sim(s,t)=a_{m,n}$ se fait en temps $O(nm)$ par application de la formule de récurrence pour des préfixes de plus en plus longs.

Algorithme de programmation dynamique :

```
pour i←0 à m
   a_{i,0}←i×g
fin pour
pour j←0 à n
   a_{0,j}←j×g
fin pour
pour i←1 à m
   pour j←1 à n
      a_{i,j}←max( a_{i-1,j}+g , a_{i-1,j-1}+p(s_i,t_j) , a_{i,j-1}+g )
   fin pour
fin pour
```

Exemple :

			t		
a		A	G	C	
	0	-2	-4	-6	
A	-2	1	-1	-3	
A	-4	-1	0	-2	
A	-6	-3	-2	-1	
C	-8	-5	-4	-1	

s

$sim(s,t)$

Remarque : plusieurs alignements peuvent être optimaux

Lorsque $sim(s,t)$ est connu, la matrice a est utilisée pour reconstituer un alignement optimal :

- à partir de la position m,n, on se « déplace » vers la position $0,0$
- à chaque étape, parmi les 3 possibilités, on retrouve celle qui avait conduit au meilleur score
  Si plusieurs choix possibles ⇒plusieurs alignements optimaux
- l'alignement est donc construit de la droite vers la gauche.

Reprenons l'exemple :

		t			
	a		A	G	C
		0	-2	-4	-6
A		-2	1	-1	-3
A		-4	-1	0	-2
A		-6	-3	-2	-1
C		-8	-5	-4	-1

s

3 alignements possibles pour une similarité de -1 :

AAAC	AAAC	AAAC
AG-C	A-GC	-AGC

Le temps de construction de l'alignement est en $O(n+m)$

Remarques :

On dira que l'algorithme de programmation dynamique a une complexité en temps quadratique car souvent on recherche les similarités entre des séquences qui ont approximativement la même longueur n. La complexité est alors $O(n^2)$.

Si les deux séquences ont la même longueur n et si on sait qu'elle sont très semblables, on peut restreindre le calcul des valeurs de a dans une bande de largeur k autour de la diagonale principale. La complexité en temps est alors $O(k.n)$.

Similarité et distance

La notion de distance peut également être utilisée pour aligner des séquences.

La distance *dist* entre deux séquences doit posséder les 3 propriétés suivantes :

1. $dist(s,s)=0$ pour tout $s \in A^*$
2. $dist(s,t)=dist(t,s)$ pour tout $s,t \in A^*$ *(symétrie)*
3. $dist(s,t) \leq dist(s,z)+dist(z,t)$ pour tout $s,t,z \in A^*$ *(inégalité triangulaire)*

La distance $dist(s,t)$ peut-être vue comme la « quantité » minimale d'opérations élémentaires à appliquer à s pour obtenir t.

Les opérations autorisées sont :

- la substitution d'un symbole par un autre → *mismatch pour la similarité*
- la suppression ou l'insertion d'un symbole → *gap pour la similarité*

Soit $c(a,b)>0$ le coût de substitution de a par b, avec $a,b \in A$, $a \neq b$ et soit $h>0$ le coût d'insertion ou de suppression d'un symbole.

Toute série d'opérations transformant s en t correspond à un alignement de s avec t.

Le coût de l'alignement est la somme des coûts des opérations correspondantes

Exemple : si $c(a,b)=1$ pour $a \neq b$ et $h=2$, alors le coût de l'alignement suivant est 3 :

```
GA-CGGATTAG
GATCGGAATAG
```

La distance entre s et t, $dist(s,t)$, est le coût de l'alignement de coût minimal.

Propriété:

Soit c le coût de substitution et h le coût d'insertion/suppression.
Soit $p(a,b)=M-c(a,b)$ et $g=-h+M/2$, avec M une constante, les scores des paires alignées et le score de gap.

Alors $sim(s,t)+dist(s,t)=M/2\ (m+n)$

L'alignement optimal pour l'approche par distance est donc également un alignement optimal pour l'approche par similarité.

Alignement local

L'alignement local entre s et t est l'alignement, entre un facteur de s et un facteur de t, de score maximal.

Exemple : s=A**TAGCAG**G et t=TC**TAGTCAG**TC

Alignement local :

```
TAG-CAG   : facteur de s
TAGTCAG   : facteur de t
```

L'algorithme est une variation de l'algorithme de programmation dynamique : la valeur $a_{i,j}$ est le score du meilleur alignement entre un suffixe de $s_{1..i}$ et un suffixe de $t_{1..j}$. Dès lors :

- les valeurs $a_{i,0}$ et $a_{0,j}$ sont initialisées avec des 0
- pour chaque entrée (i,j), il existe toujours un alignement, de score 0 entre le suffixe vide de $s_{1..i}$ et le suffixe vide de $t_{1..j}$. La valeur $a_{i,j}$ ne sera donc jamais négative $\Rightarrow$ quatre alternatives pour la valeur $a_{i,j}$:

$$a_{i,j} = \max \begin{cases} a_{i,j-1}+g \\ a_{i-1,j-1}+p(s_i,t_j) \\ a_{i-1,j}+g \\ 0 \end{cases}$$

La valeur maximale $a_{i,j}$ détermine alors le score de l'alignement local recherché. Les positions i et j sont les positions finales des facteurs de s et de t qui s'alignent au mieux.

Il suffit alors de reconstituer l'alignement à partir de la position (i,j) comme dans le cas de l'alignement global jusqu'à ce qu'une valeur $a_{i',j'}=0$ soit rencontrée. Le facteur $s_{i'+1..i}$ s'aligne alors avec $t_{j'+1..j}$.

Exemple : s=A**TAGCAG**G et t=TC**TAGTCAG**TC

		T	C	T	A	G	T	C	A	G	T	C
	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	1	0	0	0	1	0	0	0
T	0	1	0	**1**	0	0	1	0	0	0	1	0
A	0	0	0	0	**2**	0	0	0	1	0	0	0
G	0	0	0	0	0	**3**	**1**	0	0	1	0	0
C	0	0	1	0	0	1	2	**2**	0	0	0	1
A	0	0	0	0	1	0	0	1	**3**	1	0	0
G	0	0	0	0	0	2	0	0	1	**4**	2	0
G	0	0	0	0	0	1	1	0	0	2	3	1

Alignement local : **TAG-CAG** : facteur de s
TAGTCAG : facteur de t

Alignement semi-global

Souhait : l'alignement
```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
```

devrait être préféré à
```
GAGCACTTGGATTCTCGG
CAGC-----G-T----GG
```

L'idée est de ne pas pénaliser les gaps situés aux extrémités des séquences.

Comment ne pas pénaliser les gaps en début des séquences ?

Il suffit d'initialiser les $a_{i,0}$ et $a_{0,j}$ avec des 0 plutôt qu'avec les pénalités de gaps.

Comment ne pas pénaliser les gaps en fin des séquences ?

Après avoir calculé les valeurs de la matrice a, il suffit de choisir la valeur maximale parmi les $a_{m,j}$ et les $a_{i,n}$.

Après avoir trouvé cette valeur maximale, on reconstruit l'alignement en « remontant » les paires alignées jusqu'à une valeur $a_{i',0}$ ou $a_{0,j'}$.

Exemple : s = CAGCACTTGGATTCTCGG et t = CAGCGTGG

			C	A	G	C	G	T	G	G
		0	0	0	0	0	0	0	0	0
C	|	0	1	-1	-1	1	-1	-1	-1	-1
A	|	0	-1	2	0	-1	0	-2	-2	-2
G	|	0	-1	0	3	1	0	-1	-1	-1
C	|	0	**1**	-1	1	4	2	0	-2	-2
A	|	0	-1	**2**	**0**	2	3	1	-1	-3
C	|	0	1	0	1	**1**	1	2	0	-2
T	|	0	-1	0	-1	0	**0**	2	1	-1
T	|	0	-1	-2	-1	-2	-1	**1**	1	0
G	|	0	-1	-2	-1	-2	-1	-1	**2**	2
G	|	0	-1	-2	-1	-2	-1	-2	0	**3**
A	|	0	-1	0	-2	-2	-3	-2	-2	1
T	|	0	-1	-2	-1	-3	-3	-2	-3	-1
T	|	0	-1	-2	-3	-2	-4	-2	-3	-3
C	|	0	1	-1	-3	-2	-3	-4	-3	-4
T	|	0	-1	0	-2	-4	-3	-2	-4	-4
C	|	0	1	-1	-1	-1	-3	-4	-3	-5
G	|	0	-1	0	0	-2	0	-2	-3	-2
G	|	0	-1	-2	1	-1	-1	-1	-1	-2

L'alignement semi-global optimal :

CAG**CA**-**C**T**TGG**ATTCTCGG
CAG**C**G**TGG**

En général, 3 situations possibles :

Fonctions générales de pénalité de gap

D'un point de vue biologique, un gap de k symboles consécutifs correspond à un seul événement mutationnel. Il ne devrait donc pas être pénalisé de la même façon que k gaps individuels.

Soit $w(k)$ la pénalité attribuée à un gap de k symboles.

Remarque : jusqu'à présent, on avait $w(k)=|k\times g|$.

Comment trouver l'alignement de score maximal ?

Le score d'un alignement n'est plus additif : on ne peut plus couper l'alignement en des positions arbitraires et supposer que le score global est la somme des scores des parties.

On décompose un alignement en blocs de paires alignées.

3 types de blocs :

1. une paire de symboles de A
2. une série maximale de symboles de t alignés avec des espaces dans s
3. une série maximale de symboles de s alignés avec des espaces dans t

Pénalité des blocs de type 1 : $p(a,b)$ avec a et b les symboles alignés
Pénalité des blocs de type 2 et 3 : $-w(k)$ où k est la longueur du gap

L'alignement

```
AAC---AATTCCGACTAC
ACTACCT------CGC--
```

se décompose en 10 blocs :

```
A|A|C|---|A|ATTCCG|A|C|T|AC
A|C|T|ACC|T|------|C|G|C|--
```

Le score d'un alignement est additif si l'alignement est décomposé selon les frontières de blocs

Un bloc de type 2 ou 3 ne peut jamais être suivi d'un bloc de même type !

Pour chaque couple (i,j) on maintient 3 scores de meilleur alignement entre $s_{1..i}$ et $t_{1..j}$:

- le score du meilleur alignement se terminant par un bloc de type 1 : $a_{i,j}$
- le score du meilleur alignement se terminant par un bloc de type 2 : $b_{i,j}$
- le score du meilleur alignement se terminant par un bloc de type 3 : $c_{i,j}$

Valeurs initiales de la première ligne et première colonne de a,b et c :

$$\begin{cases} a_{0,0} = 0 \\ b_{0,j} = -w(j) \text{ pour } 0<j\leq n \\ c_{i,0} = -w(i) \text{ pour } 0<i\leq m \\ \text{pour les autres indices, on donne la valeur } -\infty \text{ à } a,b \text{ et } c \text{ (pas de sens)} \end{cases}$$

Les relations de récurrence sont :

$$a_{i,j} = p(s_i,t_j) + \max \begin{cases} a_{i-1,j-1} \\ b_{i-1,j-1} \\ c_{i-1,j-1} \end{cases}$$

$$b_{i,j} = \max \begin{cases} a_{i,j-k} - w(k) \text{ pour } 1\leq k\leq j \\ c_{i,j-k} - w(k) \text{ pour } 1\leq k\leq j \end{cases}$$

$$c_{i,j} = \max \begin{cases} a_{i-k,j} - w(k) \text{ pour } 1\leq k\leq i \\ b_{i-k,j} - w(k) \text{ pour } 1\leq k\leq i \end{cases}$$

On a alors $sim(s,t) = \max (a_{m,n} , b_{m,n} , c_{m,n})$

La reconstruction de l'alignement optimal se fait comme pour l'alignement classique.

Complexité en temps de l'alignement : $O(mn^2 + m^2n)$ *(« cubique »)*

Fonctions de gap affines

L'utilisation d'une fonction de pénalité de gap moins générale ne permettrait-elle pas de réduire la complexité en temps de l'algorithme d'alignement ?

Soit $w(k)=h+g\times k$ la pénalité attribuée à un gap de k symboles (avec $h,g>0$) :
- le premier espace d'un gap coûte $h+g$
- les espaces suivants coûtent g

L'algorithme de programmation dynamique doit faire une distinction entre le premier espace d'un gap et les suivants.

Pour chaque couple (i,j) on maintient 3 scores de meilleur alignement entre $s_{1..i}$ et $t_{1..j}$:

- le score du meilleur alignement se terminant par la paire $\begin{matrix} s_i \\ t_j \end{matrix}$: $a_{i,j}$
- le score du meilleur alignement se terminant par la paire $\begin{matrix} - \\ t_j \end{matrix}$: $b_{i,j}$
- le score du meilleur alignement se terminant par la paire $\begin{matrix} s_i \\ - \end{matrix}$: $c_{i,j}$

Valeurs initiales de la première ligne et première colonne de a,b et c :

$$\begin{cases} a_{0,0} = 0 \\ a_{i,0} = -\infty \text{ pour } 1 \le i \le m \text{ (pas de sens)} \\ a_{0,j} = -\infty \text{ pour } 1 \le j \le n \text{ (pas de sens)} \end{cases}$$

$$\begin{cases} b_{i,0} = -\infty \text{ pour } 1 \le i \le m \text{ (pas de sens)} \\ b_{0,j} = -(h+gj) \text{ pour } 1 \le j \le n \end{cases}$$

$$\begin{cases} c_{i,0} = -(h+gi) \text{ pour } 1 \le i \le m \\ c_{0,j} = -\infty \text{ pour } 1 \le j \le n \text{ (pas de sens)} \end{cases}$$

Les relations de récurrence permettent de faire la distinction entre le premier espace d'un gap et les suivants :

$$a_{i,j} = p(s_i,t_j) + \max \begin{cases} a_{i-1,j-1} \\ b_{i-1,j-1} \\ c_{i-1,j-1} \end{cases}$$

$$b_{i,j} = \max \begin{cases} -(h+g) + a_{i,j-1} \\ -g + b_{i,j-1} \\ -(h+g) + c_{i,j-1} \end{cases}$$

$$c_{i,j} = \max \begin{cases} -(h+g) + a_{i-1,j} \\ -(h+g) + b_{i-1,j} \\ -g + c_{i-1,j} \end{cases}$$

On a alors $sim(s,t) = \max (a_{m,n} , b_{m,n} , c_{m,n})$

La reconstruction de l'alignement optimal se fait comme pour l'alignement classique.

Complexité en temps de l'alignement :
$O(mn)$

Comparaison de plus de 2 séquences

Soit $s_1,s_2,...,s_k \in A^*$. On veut évaluer et localiser les similitudes entre les k séquences.
Souvent en pratique, $k \approx 10$.

Exemple :

On possède les séquences des protéines qui possèdent la même fonction dans des espèces différentes. On désire voir où ces séquences sont similaires et où elles diffèrent.

Un alignement multiple correspond à :
- ajouter des espaces dans certaines séquences pour les amener toutes à la même longueur
- aligner les chaînes ainsi obtenues verticalement les unes en dessous des autres
- aucune colonne ne peut contenir que des espaces

Exemple :

```
MQPILLL
MLR-LL-
MK-ILLL
MPPVLIL
```


Score d'un alignement multiple :

Score additif : le score de l'alignement α est la somme des scores de ses colonnes.

Comment calculer le score d'une colonne ?

Imaginons un tableau à k dimensions $p(l_1,l_2,...l_k)$, chaque symbole l_i pouvant être une lettre de A ou un espace (-) $\Rightarrow (\#A+1)^k-1$ entrées à envisager !

Propriétés imposées :

- La fonction de score de colonne doit être indépendante de l'ordre des arguments. Par exemple, $p(\mathtt{I},\mathtt{-},\mathtt{I},\mathtt{V})=p(\mathtt{I},\mathtt{I},\mathtt{-},\mathtt{V})$.
- La fonction doit récompenser la présence de symboles identiques et pénaliser la présence de mismatchs ou de gaps.

Score de colonne SP (*sum-of-pairs*) :

Somme des scores de toutes les paires composant la colonne :

$$SP\text{-}score(\mathtt{I},\mathtt{-},\mathtt{I},\mathtt{V})=p(\mathtt{I},\mathtt{-})+p(\mathtt{I},\mathtt{I})+p(\mathtt{I},\mathtt{V})+p(\mathtt{-},\mathtt{I})+p(\mathtt{-},\mathtt{V})+p(\mathtt{I},\mathtt{V})$$

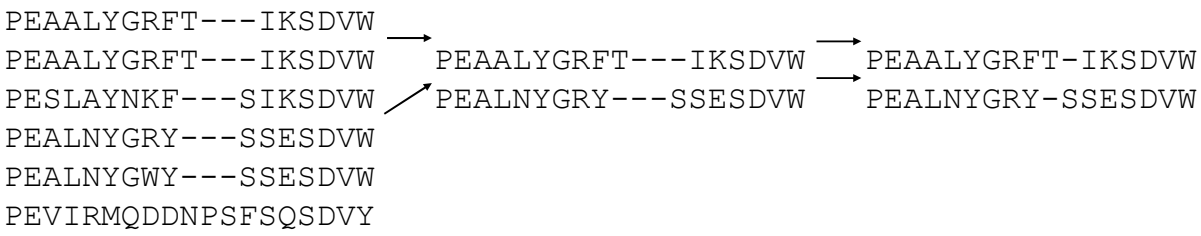
avec $p(a,b)$ le score de la paire de symboles a et b, avec $a,b \in A \cup \{-\}$

Remarques :

L'alignement multiple induit $\frac{k(k-1)}{2}$ alignements de 2 séquences.
Les alignements induits peuvent posséder des paires alignées contenant deux -.

Si $p(-,-)=0$, alors $SP\text{-}score(\alpha)=\sum_{i<j} score(\alpha_{ij})$ où α_{ij} est l'alignement de s_i avec s_j induit par l'alignement multiple α .

Exemple : α α_{24}



La complexité en temps de l'évaluation du score d'une colonne est $O(\frac{k(k-1)}{2})=O(k^2)$.

D'autres fonctions de score moins coûteuses sont parfois utilisées, par exemple le comptage du nombre de symboles autres que - par colonne : temps $O(k)$.

Alignement par programmation dynamique : méthode exacte

L'alignement optimal est celui qui maximise le score *SP-score*.

Algorithme de programmation dynamique : même méthode que pour 2 séquences.

Supposons k séquences de longueur n.

Tableau a à k dimensions, avec $(n+1)$ indices pour chaque dimension :
$a_{i1,i2,...ik}$ est le meilleur score d'alignement des préfixes $s_{1:1..i1}$, $s_{2:1..i2}$, ...$s_{k:1..ik}$

- $(n+1)^k$ valeurs à calculer. Le score final d'alignement sera $a_{n,n,...n}$
- chaque valeur dépend de 2^k-1 valeurs déjà calculées
- l'évaluation du score de chacune de ces 2^k-1 valeurs prend un temps $O(k^2)$ sauf si une fonction de score plus simple est utilisée.

⇒Complexité en temps globale : $O(k^2 2^k n^k)$!!!

Problème NP-complet.

Problème technique : on peut rarement définir la dimension d'un tableau à l'exécution ⇒définir sa propre méthode d'accès ⇒complexité multipliée par k !

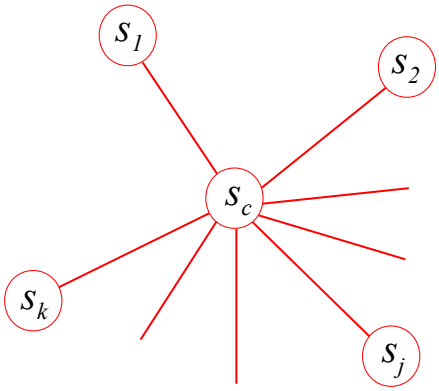
Alignement en étoile : méthode approchée

Une séquence « pivot » s_c est choisie parmi les k séquences.

Tous les alignements 2 à 2 de s_c *et* s_i, avec $i \neq c$ sont construits.

L'alignement multiple α est une « synthèse » de ces alignements de 2 séquences.

α est tel que ses alignements induits α_{ij} sont optimaux mais α n'est pas forcément optimal



Si les k séquences sont de longueur n, alors
le calcul des alignements prend un temps $O(k\,n^2)$.

Construction de l'alignement multiple :
- initialement : alignement de s_1 avec s_c
- la fusion de l'alignement α_{ci} avec l'alignement multiple actuel est réalisé en respectant les paires alignées dans α_{ci} :
  Les gaps dans α_{ci} induisent des gaps dans l'alignement multiple si - est dans s_c.
  Les - dans s_c dans l'alignement multiple impose des - dans s_i.

Si la longueur maximale de l'alignement multiple est l, alors
le temps de chaque fusion est en $O(k\,l)$.

La complexité en temps totale est donc $O(kn^2+k^2l)$.

Le temps éventuel de calcul du score final est en $O(k^2l)$.

Comment choisir la séquence pivot s_c ?

- les essayer toutes comme pivot et choisir celle qui donne le meilleur score
- choisir celle qui maximise $\Sigma_{i \neq c} sim(s_i,s_c)$.

Exemple :

s_1 = ATTGCCATT
s_2 = ATGGCCATT
s_3 = ATCCAATTTT
s_4 = ATCTTCTT
s_5 = ACTGACC

sim	s_1	s_2	s_3	s_4	s_5
s_1		7	-2	0	-3
s_2	7		-2	0	-4
s_3	-2	-2		0	-7
s_4	0	0	0		-3
s_5	-3	-4	-7	-3	

Séquence pivot : s_1

Initialement : alignement de s_1 et s_2 :

```
ATTGCCATT
ATGGCCATT
```

On ajoute ensuite l'alignement de s_1 et s_3 :

```
ATTGCCATT--          ATTGCCATT--
ATC-CAATTTT    ⇒     ATGGCCATT--
                     ATC-CAATTTT
```

On ajoute ensuite l'alignement de s_1 et s_4 :

```
ATTGCCATT            ATTGCCATT--
ATCTTC-TT      ⇒     ATGGCCATT--
                     ATC-CAATTTT
                     ATCTTC-TT--
```

On ajoute enfin l'alignement de s_1 et s_5 :

```
ATTGCCATT            ATTGCCATT--
ACTGACC--      ⇒     ATGGCCATT--
                     ATC-CAATTTT
                     ATCTTC-TT--
                     ACTGACC----
```


Criblage de banques (pour information)

Le criblage de banques consiste à rechercher, dans un très grand ensemble de séquences (une banque de séquences) les similitudes avec une séquence « *query* » donnée.

Vu la quantité d'information à traiter, les algorithmes doivent être très rapides. La programmation dynamique n'est pas adaptée.

Les algorithmes utilisés sont composés d'heuristiques qui visent à localiser des occurrences exactes et à en déduire des zones similaires.

BLAST recherche des segments, de même longueur, similaires dans la séquence query et dans la séquence de la banque.
Recherche des points d'ancrage (paires de courts facteurs présents dans la séquence query et dans la séquence de la banque) qui sont alors étendus dans les 2 directions pour réaliser un alignement local sans gap.

FAST crée les listes des positions des k-uples (typiquement, pour les protéines, $k=1$ ou 2) de la séquence query et pour chaque séquence de la banque. Les occurrences communes constituent des suites de paires de k-uples alignés qui sont jointes si elles sont proches.

Références

- M. Crochemore et W. Rytter, Text Algorithms, *Oxford University Press*, 1994
- D. Gusfield, Algorithms on Strings, Trees and Sequences, *Cambridge University Press*, 1997
- J. Setubal et J. Meidanis, Introduction to Computational Molecular Biology,*PWS Publishing Company,* 1997
- G.A. Stephen, String Searching Algorithms, *World Scientific*, 1994