Recherche exacte de motifs

Nadia El-Mabrouk

Problème
Recherche exacte d'un seul motif

- Σ: Alphabet
- $T = t_1\ t_2 \ldots\ t_n$: Texte de taille n
- $P = p_1\ p_2 \ldots\ p_m$: Mot de taille m , n >> m

Trouver les positions de toutes les occurrences exactes de P dans T

P = GCG

	1	2	3	4	**5**	6	**7**	8	9	10	11	12	**13**	14	15	16	17	18
T:	**A**	**G**	**C**	**C**	**G**	**C**	**G**	**C**	**G**	**T**	**C**	**C**	**G**	**C**	**G**	**T**	**G**	**C**
					G	C	G											
							G	C	G				G	C	G			

Problème
Recherche exacte multiple

- Σ: Alphabet
- $T = t_1\ t_2 \ldots\ t_n$: Texte de taille n
- *Ensemble* $\mathcal{P} = \{P_1, P_2, \ldots P_l\}$ de mots de Σ

Trouver les positions de toutes les occurrences exactes de tous les mots de $\mathcal{P}$ dans T

P = {GCG, CGC, CCG}

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T:	**A**	**G**	**C**	**C**	**G**	**C**	**G**	**C**	**G**	**T**	**C**	**C**	**G**	**C**	**G**	**T**	**G**	**C**
					G	C	G											
							G	C	G				G	C	G			
			C	C	G	C	G	C			C	C	G					
				C	G	C												

Intérêts de la recherche exacte

- Utilitaire de base pour la manipulation de textes. Aussi important que le tri ou opérations arithmétiques de base.
- Applications courantes: utilitaires UNIX comme grep; outils de recherche web; recherche dans les cathalogues de bibliothèques ou revues électroniques…
- Utilitaire de base pour la recherche de motifs biologiques: recherche approchée (BLAST, FASTA...), recherche de répétitions, alignement multiple…

Exemple d'application biologique:

Utilisation du transcriptome pour annoter le génome.

- Transcriptome: Ensemble des ARN messagers représentant les gènes exprimés dans une lignée cellulaire donnée.
- Technologie HTS « *high-throughput screening* » permet de séquencer des segments de chaque transcrit.
- Retrouver leur position dans le génome permet une annotation des gènes fonctionnels.
- Pertinent seulement s'il existe une seule occurrence de chaque marqueur dans le génome.
- Plus les marqueurs sont longs, plus ils sont spécifiques.
- Avec la technologie HTS, possible de séquencer des marqueurs de taille ~36 en grande quantité.

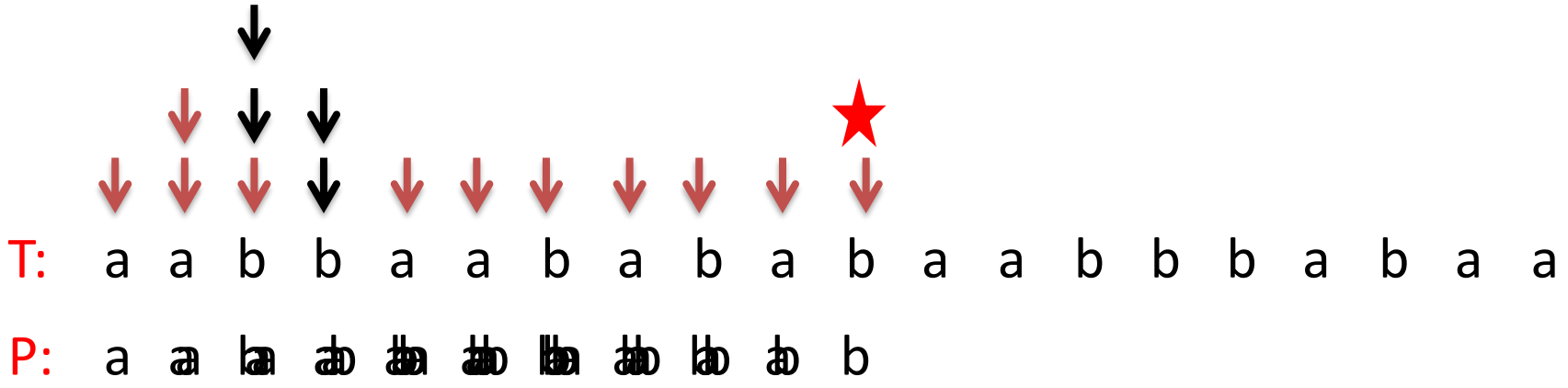
Pourquoi des algorithmes rapides? Dans notre exemple

- Génomes de très grande taille: ~10^6 pour une bactérie (*E. coli*: 4.6Mb) à 10^{11} pour certains poissons. Homme: 3.2 Gb.
- Des millions de « marqueurs » séquencés d'un transcriptome.

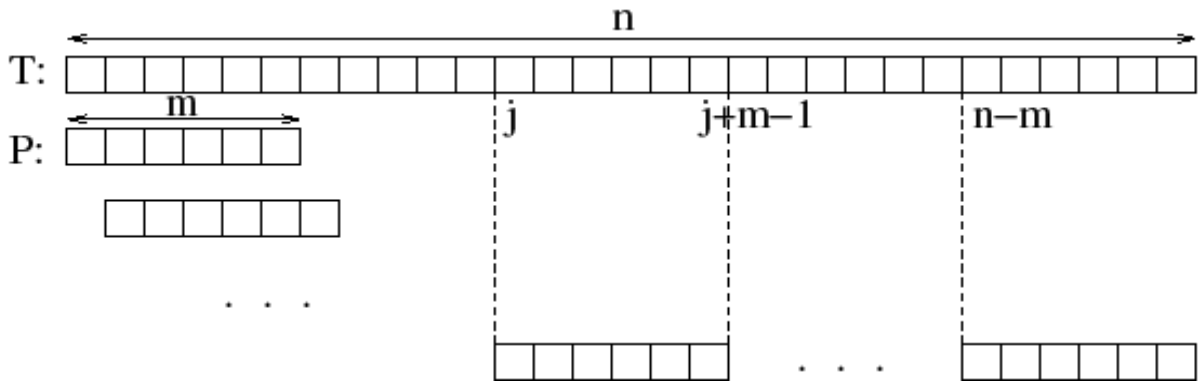
Pourquoi des algorithmes rapides? Dans le cas général

- Catalogues électroniques gigantesques;
- Banques de données biologiques croissante de façon exponentielle.
- Recherche exacte souvent utilisée comme étape de filtrage dans des logiciels complexes. Doit être le plus rapide possible.

Recherche exacte- Algorithme naïf



Recherche exacte- Algorithme naif




```
Algorithme recherche-naive (T, n, P  m)
    Pour  j = 0 à n − m Faire
            i := 0;
            Tant que (T[j+i] = P[i] et i < m) i := i + 1;
            Si i = m Signaler une occurrence de P
    Fin Pour
```

Complexité: O(mn) dans le pire des cas.

Pour un alphabet suffisamment grand, nombre moyen de comparaisons O(n)

Optimisations de l’algorithme naïf

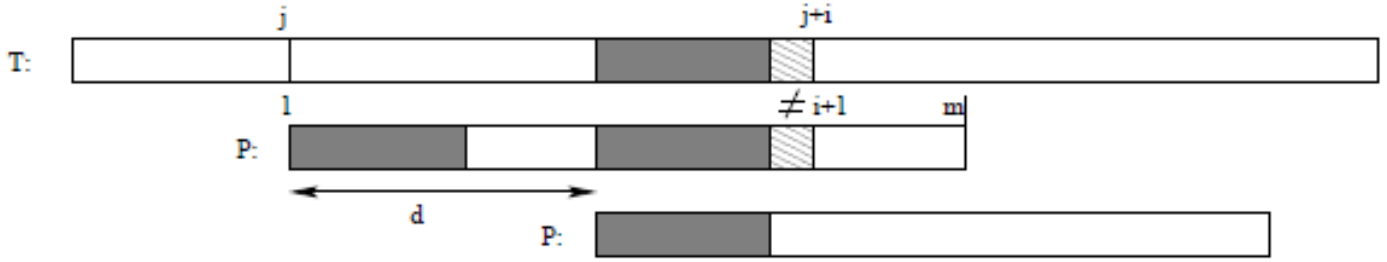
- Ne pas recomparer les mêmes caractères d’une étape à l’autre;
- Décaler le mot de plus d’un caractère à la fois;
- Éviter de considérer certaines parties du texte.

Approche Morris-Pratt (1970)

- À chaque étape, décalage de *P* de plus d'un caractère.
- Index *j* sur le texte jamais décrémenté
- Les décalages ne dépendent que de *P*

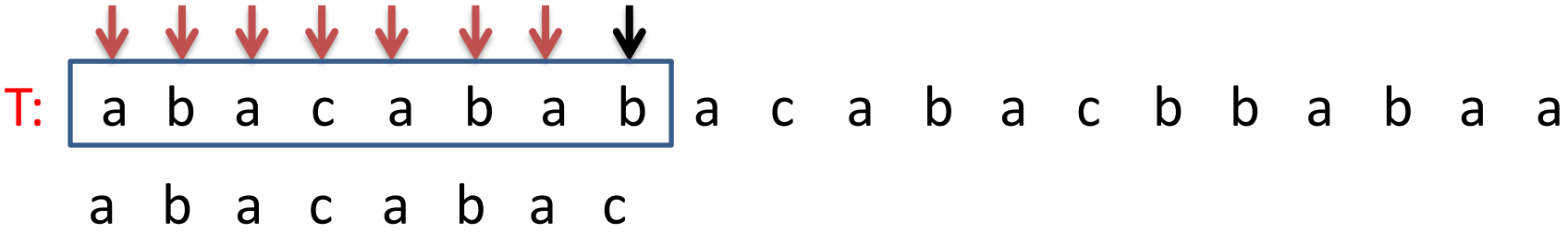
Approche Morris-Pratt (1970)

Si $P[1..i] = T[j..j+i-1]$ et $p_{i+1} \neq t_{j+i}$, décaler P pour que le plus long préfixe de $p_1 \cdots p_i$ qui est aussi un suffixe, soit aligné avec le facteur de T finissant à la position $j+i-1$. Reprendre les comparaisons entre $p_{i+1}p_{i+2}\cdots$ et $t_{j+i}t_{j+i+1}\cdots$



- Bord d'un mot *u* : facteur de *u*, à la fois préfixe et suffixe de *u.*
- Bord maximal de *u* : Plus long bord de *u*.

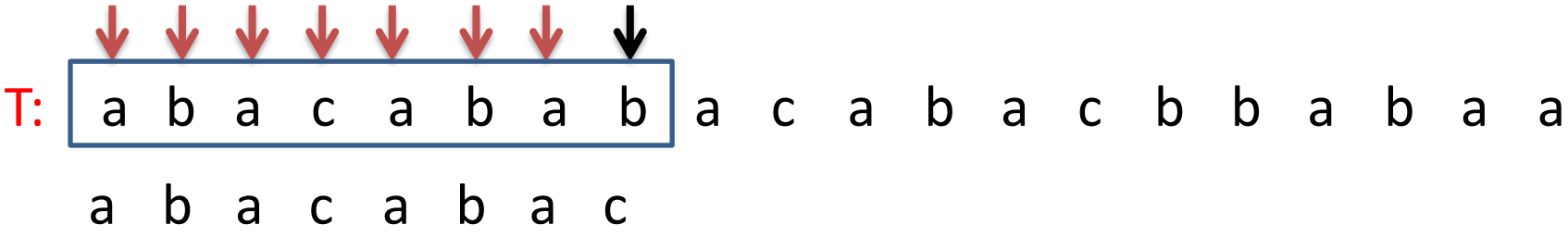
Algorithme MP



a b a c a b a x

a b a c a b a c

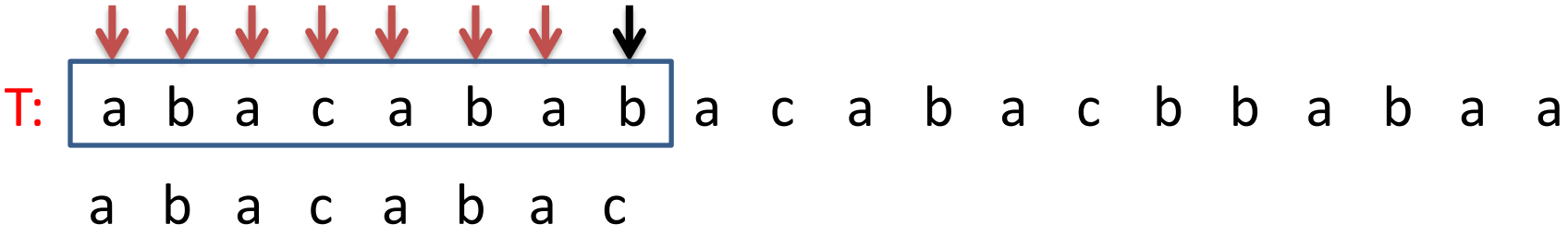
Algorithme MP



a b a c a b a x

a b a c a b a c

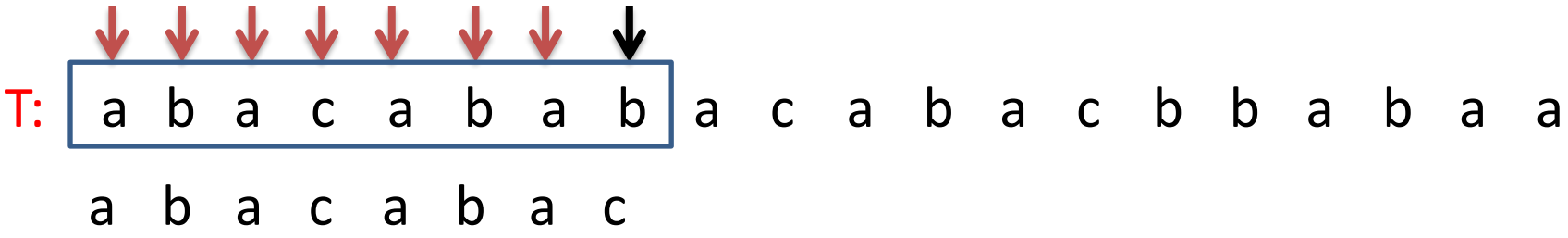
Algorithme MP



a b a c a b a x

a b a c a b a c

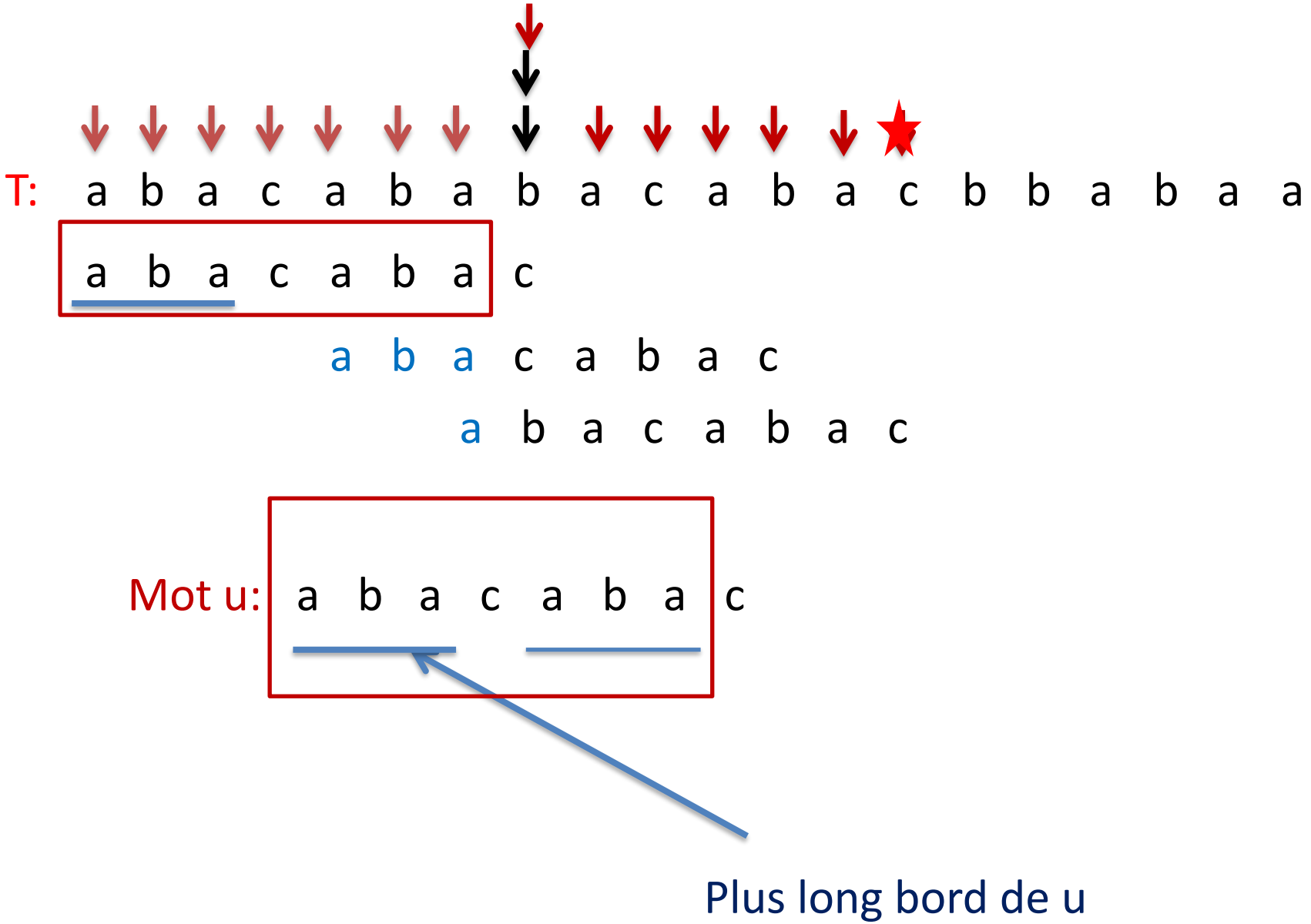
Algorithme MP



a b a c a b a x

a b a c a b a c

Algorithme MP



Approche Morris-Pratt(1977)

- À chaque étape, décalage de *P* de plus d'un caractère.
- Index *j* sur le texte jamais décrémenté
- Les décalages ne dépendent que de *P* ➔ les calculer au cours d'une phase de prétraitement de *P*

Approche Morris-Pratt (1970)

Exemple de prétraitement :

Soit $P =$ abacabac. ε indique un bord vide.

P		a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7	8
Bord(P_i)	-	ε	ε	a	ε	a	ab	aba	abac
$\varphi(i)$	-1	0	0	1	0	1	2	3	4

```
T: a b a c a b a b a c a b a c ..............
   a b a c a b a c
         a b a c a b a c
            a b a c a b a c
               a b a c a b a c
```

Approche Morris-Pratt (1970)

On note $\varphi(i)$ = Bord (P[1,i]).

De plus, on fait suivre le dernier caractère de P d'un caractère spécial (pas dans l'alphabet du texte)

Algorithme MP :

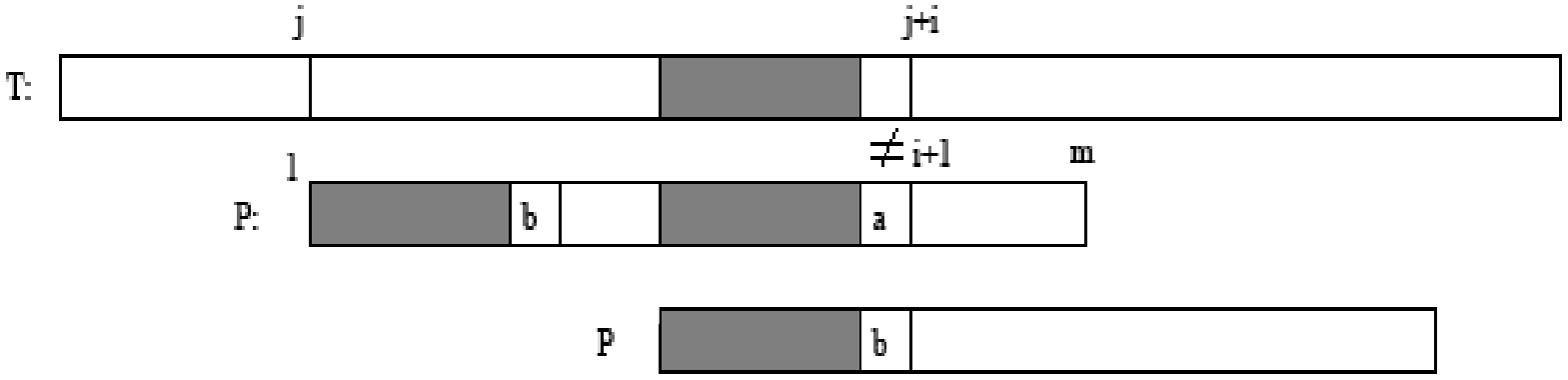
```
i = 1; j =1;
Tant que j ≤ n Faire
        Si T_j ≠ P_i alors
                Si i=1 alors j=j+1
                Sinon i = 1 + φ(i-1);
        Sinon i = i+1; j = j+1;
        Si i > m alors
                « Occurrence de P débutant à la position j-m »;
        Finsi
Fin Tant que.
```

Approche Morris-Pratt(1977)

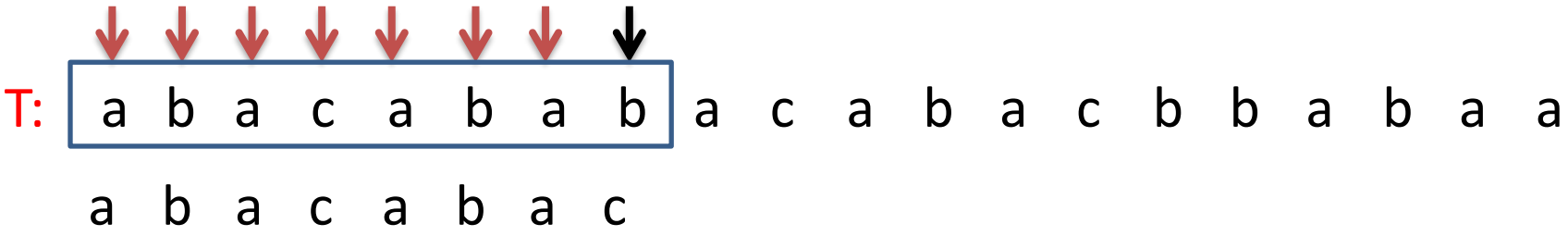
- À chaque étape, décalage de *P* de plus d'un caractère.
- Index *j* sur le texte jamais décrémenté
- Les décalages ne dépendent que de *P* ➔ les calculer au cours d'une phase de prétraitement de *P*
- |T| = n; |P| = m: Parcours du texte en O(n), prétraitement en O(m) ➔ O(n+m)

Approche Knuth-Morris-Pratt(1977)

- Bord disjoint de *u*: Bord, dont les parties préfixe et suffixe sont succédées de caractères différents.



Algorithme KMP

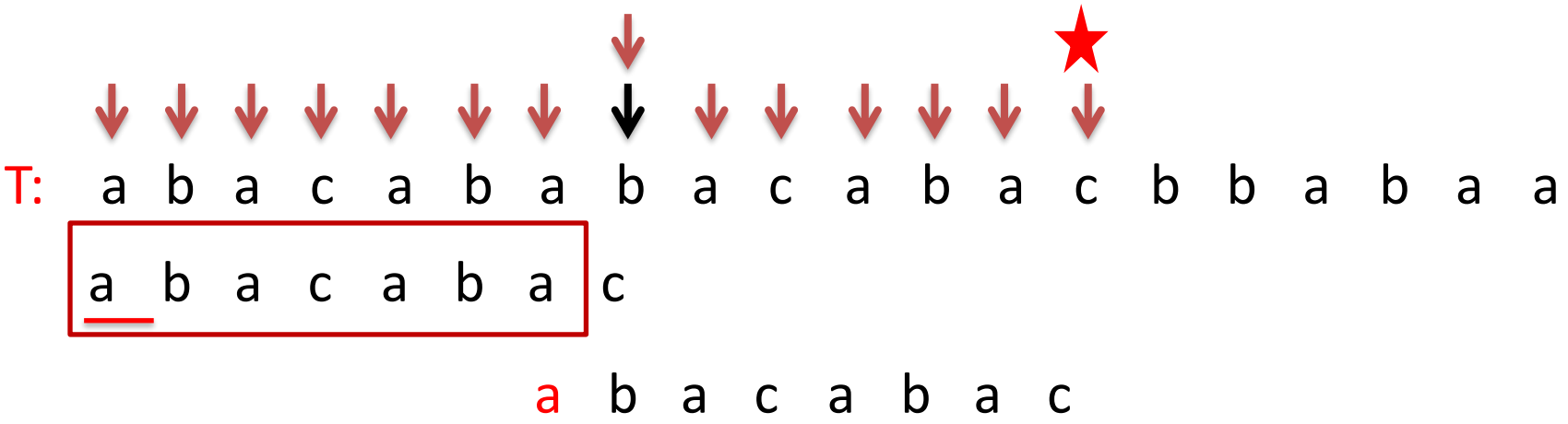


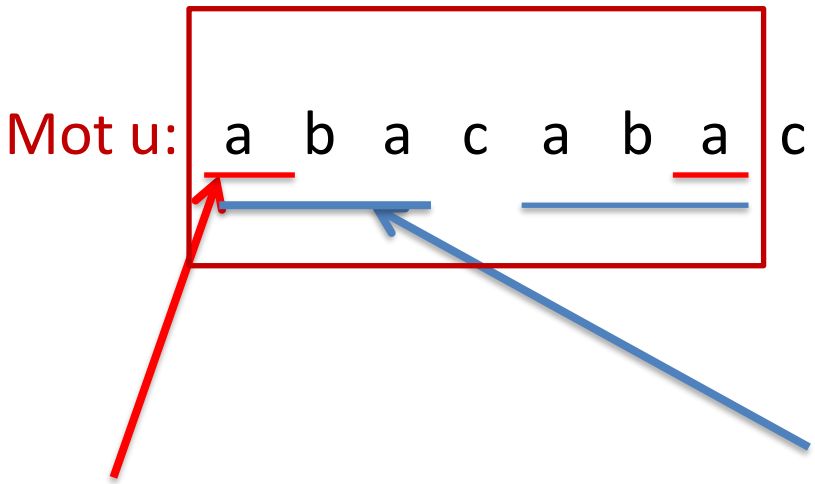
a b a c a b a x

Avec x différent de c

a b a c a b a c

Algorithme KMP





Approche KMP

Exemple de prétraitement :

P		a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7	8
DBord(P_i)	-	ε	ε	a	ε	ε	ε	a	abac
$\gamma(i)$	-1	0	0	1	0	0	0	1	4

```
T: a b a c a b a b a c a b a c ..............
   a b a c a b a c
             a b a c a b a c
                   a b a c a b a c
```

Approche KMP

Complexité :

Même complexité en temps dans le pire des cas que MP.

Il est clair que KMP est plus rapide. Une façon d'évaluer la complexité de l'algorithme est de considérer le délai, i.e. le nombre maximal de comparaisons effectuées sur un caractère du texte.

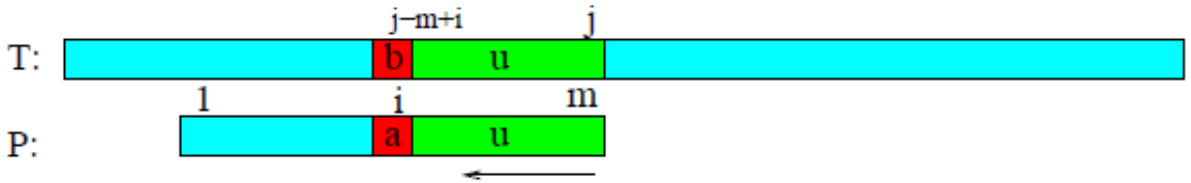
Délai de MP: peut atteindre m

Délai de KMP: ne dépasse pas $1 + \log_\phi m$, où $\phi = (1 + \sqrt{5})/2$.

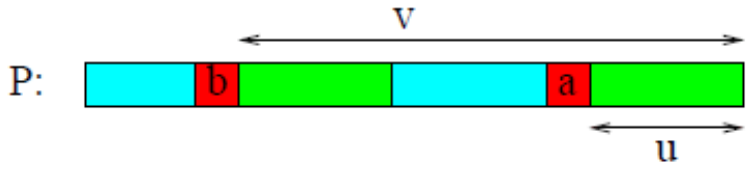
Algorithme Boyer-Moore (1977)

- Linéaire dans le pire des cas.
- Saute des caractères du texte → sous-linéaire en moyenne.
- Déroulement:
  - À chaque position j, parcourir le mot *P* de droite à gauche.
  - S'arrêter dès qu'on arrive au début de *P* (occurrence finissant à la position *j*), ou dès que les caractères comparés diffèrent.
  - Décalage de *P*.

Algorithme Boyer-Moore (1977)



bord disjoint droit d'un suffixe v de P: Bord u de v tel que u et v ne sont pas précédés du même caractère dans P.

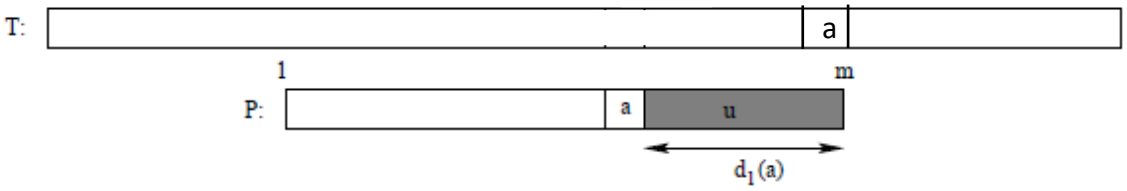


Le décalage par le bord disjoint droit consiste à décaler P de telle sorte que le bord disjoint **le plus à droite** soit aligné avec u.

Algorithme Boyer-Moore (1977)

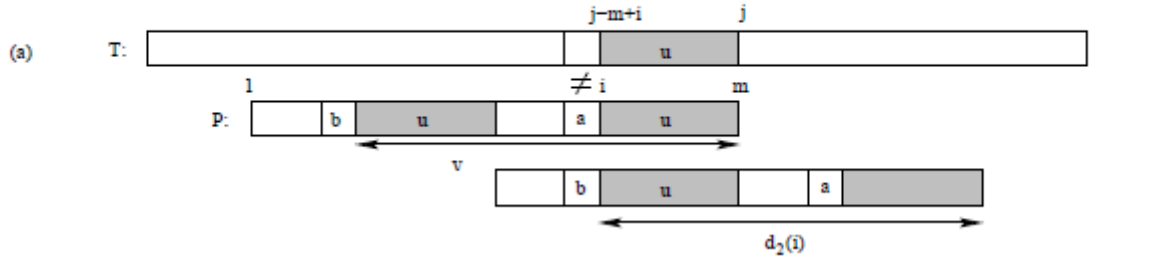
Le décalage d dépend de deux fonctions:

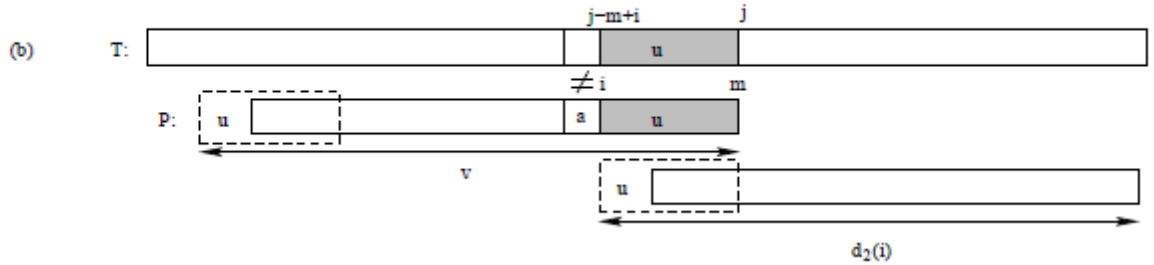
Décalage d_1:



(Note: Dans l'algo d'origine, d_1 se fait plutôt par rapport au caractère qui a causé le mismatch.)

Décalage d_2:





Algorithme Boyer-Moore (1977)

Supposons: $u = P[i+1..m] = T[j-m+i+1..j]$ et $p_i \neq t_{j-m+i}$.

- d_1: Décalage minimal pour que t_j coïncide avec un caractère de P. Pour tout $a \in \Sigma$:

$$d_1(a) = \min\{d \,/ (d = m) \text{ ou } (d = |u|,\ u \neq \varepsilon \text{ et } au \text{ est un suffixe de } P)\} .$$

- Pour tout i, $0 \leq i \leq m$, et $u = P[i+1..m]$:

$$d_2(i) = \min\{|v| \ / \ v \in V(u) \cup W(u)\} \text{ où}$$

$$V(u) = \{v \ / \ v \text{ est suffixe de } P \text{ et } u \text{ est bord disjoint droit de } v\} .$$

$$W(u) = \{v \ / \ P \text{ est suffixe de } v,\ u \text{ est bord de } v \text{ et } |v| \leq m|u|\} .$$

Le décalage effectué dépend de $d_1(T_j)$ et de $d_2(i)$

Algorithme Boyer-Moore (1977)

Algorithme BM :

$j := m$;

Tant que $j \leq n$ faire

  $i := m$; $j^* = j$

  Tant que $i > 0$ et $t_j = p_i$ faire

    $i := i - 1$; $j := j - 1$;

  Si $i = 0$ alors

    "Occurrence de P débutant à la position j";

    $j := j + d_2(i)$;

  Sinon

    $j :=$ $max\ (j^* + d_1(T_{j^*}),\ j + d_2(i))$

Algorithme Boyer-Moore (1977)

Exemple de déroulement

$P =$ aababab.

Σ	a	b	c
d_1	1	2	7

i	0	1	2	3	4	5	6	7
P		a	a	b	a	b	a	b
d_2	14	13	12	6	10	6	8	1

```
T: a a b b a a b a b a c a a b b b a b a a ......
   a a b a b a b
         a a b a b a b
                  a a b a b a b
                           a a b a b a b
```

Algorithme Boyer-Moore (1977)

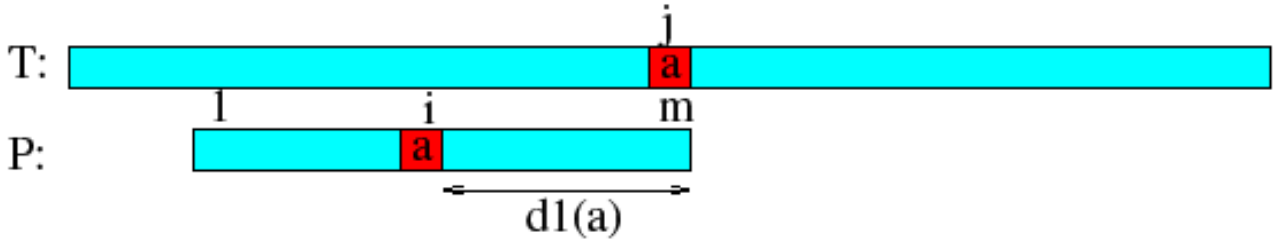
Complexité:

- Calcul de d_1 en temps $O(m + |\Sigma|)$
- Calcul de d_2 en temps $O(m)$
- Parcours du texte nécessite $O(n + rm)$ comparaisons, où r est le nombre d'occurrences de P dans T.

Algorithme d'autant plus efficace de Σ est grand. Dans ce cas, nombre de comparaisons se rapproche, dans la pratique, de n/m
$\Longrightarrow$ sous-linéaire en pratique

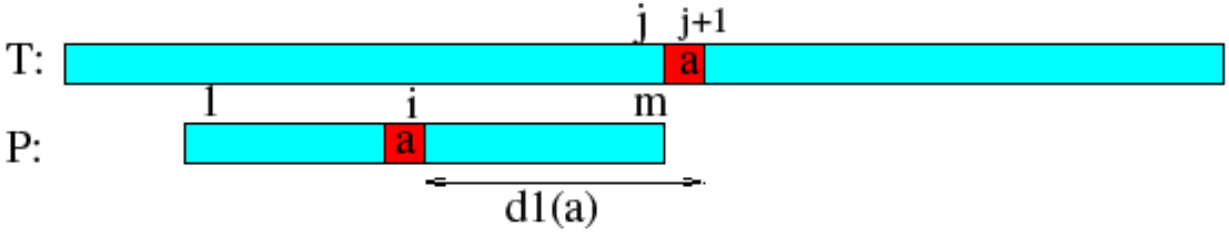
Algorithme de Horspool et Sunday

- Ne tiennent pas compte de la périodicité des mots
- Horspool: Décalage d_j dépend uniquement de t_j ; plus petit décalage nécessaire pour faire coïncider t_j avec un caractère de *P*.



Algorithme de Horspool et Sunday

- Ne tiennent pas compte de la périodicité des mots
- Sunday: Décalage d_j dépend uniquement de t_{j+1}



Algorithme de Boyer-Moore-Horspool

- Horspool avec comparaisons des caractères de P de droite à gauche.
- Performance sous-linéaire en moyenne. Dans le pire des cas, complexité en O(mn).

Optimisations supplémentaires

- Alphabet de l'ADN limité à 4 lettres → décalages pas très grands.
- Optimisations possibles: Effectuer le décalage en considérant plus qu'un caractère: q-mers (ou q-grams) (Zhu and Takaoka 1987, Baeza-Yates 1989, Kim and Shawe-Taylor 1994…)

Recherche Multiple

- Σ: Alphabet
- $T = t_1\ t_2 \ldots\ t_n$: Texte de taille n
- *Ensemble* $\mathcal{P} = \{P_1, P_2, \ldots P_l\}$ de mots de Σ

Trouver les positions de toutes les occurrences exactes de tous les mots de $\mathcal{P}$ dans T

P = {GCG, CGC, CCG}

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T:	**A**	**G**	**C**	**C**	**G**	**C**	**G**	**C**	**G**	**T**	**C**	**C**	**G**	**C**	**G**	**T**	**G**	**C**
					G	C	G											
							G	C	G				G	C	G			
				C	G	C					C	C	G					
						C	G	C										

Recherche Multiple

- Application directe des algos de recherche exacte: *O(m+l.n)* (*m* taille totale des mots, l nombre de mots)

- **Objectif**: Recherche en un seul parcours du texte : O(m+n)

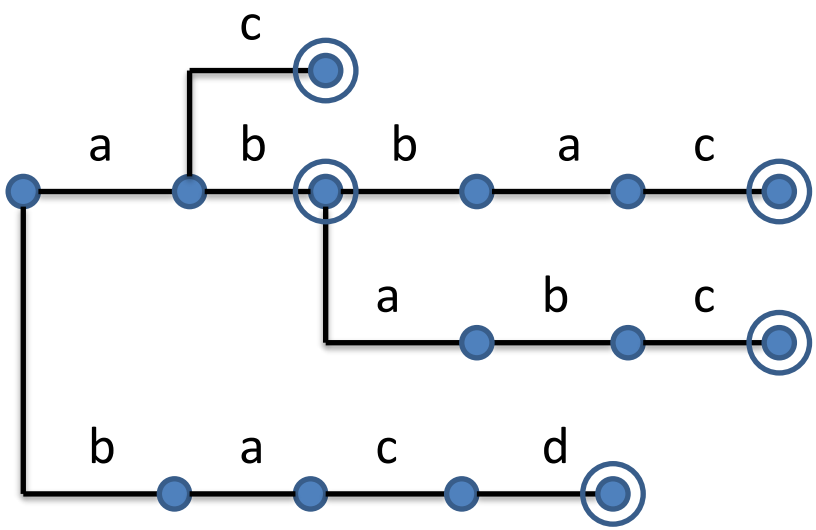
P = {GCG, CGC, CCG}

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T:	**A**	**G**	**C**	**C**	**G**	**C**	**G**	**C**	**G**	**T**	**C**	**C**	**G**	**C**	**G**	**T**	**G**	**C**
					G	C	G											
							G	C	G				G	C	G			
				C	G	C					C	C	G					
						C	G	C										

Algorithme de Aho-Corasick

- Utilisé dans le programme fgrep d'UNIX
- Basé sur la démarche KMP
- Prétraitement: Construction d'un arbre $\mathcal{K}$, arbre de Aho-Corasick, index sur les mots de $\mathcal{P}$

$\mathcal{P}$ = {abbac, ac, bacd, ababc, ab}



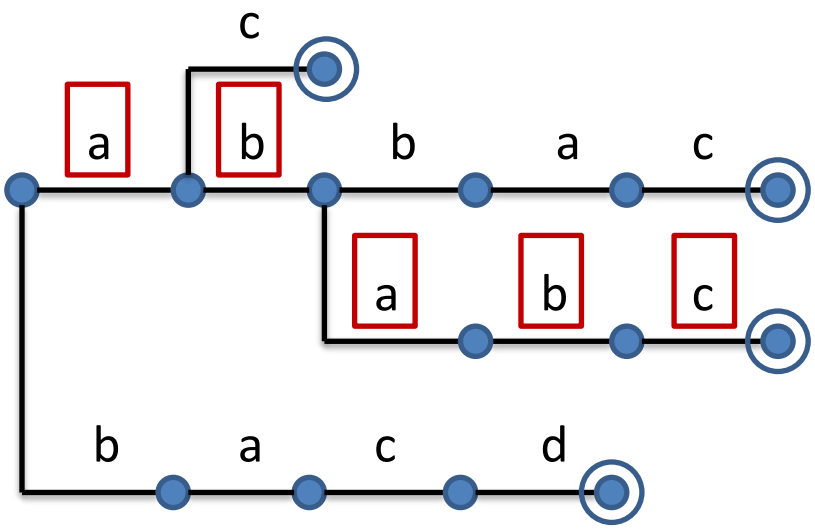
Arbre de Aho-Corasick

- Définition: L'arbre de Aho-Corasick pour un ensemble de mots $\mathcal{P}$ est un arbre enraciné orienté $\mathcal{K}$ satisfaisant les 4 contraintes suivantes:
  - Chaque arête est étiquetée par un et un seul caractère.
  - Deux arêtes sortant d'un même sommet ont des étiquettes différentes.
  - Chaque mot de p_i de $\mathcal{P}$ est associé à un sommet *v* de $\mathcal{K}$: i.e. les caractères étiquetant le chemin de la racine de $\mathcal{K}$ à *v* forment le mot p_i. De plus, chaque feuille de $\mathcal{K}$ est associée à un mot de $\mathcal{P}$.
- Construction de l'arbre en O(m) (m: taille totale des mots)

Recherche dans un texte

- Recherche naive: O(mn)

$\mathcal{P}$ = {abbac, ac, bacd, ababc}

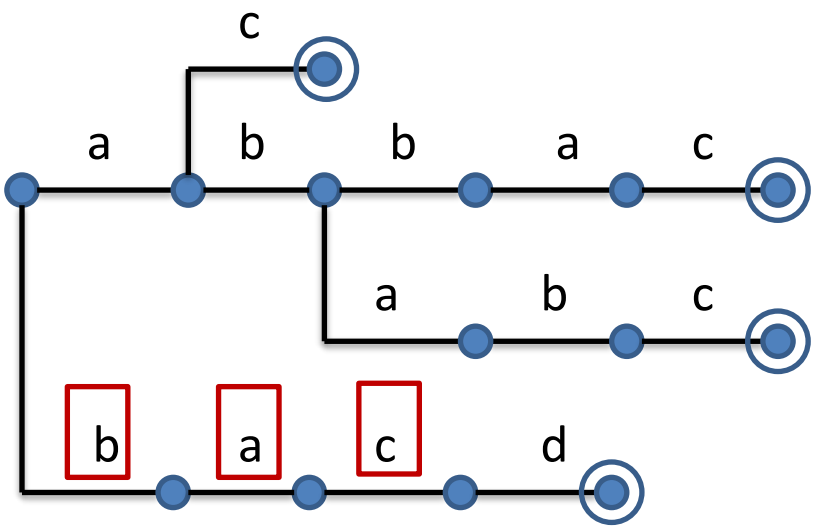


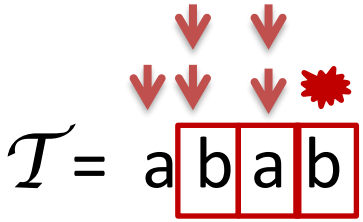
$\mathcal{T}$ = a b a b b a c d ….

Recherche dans un texte

- Recherche naive: O(mn)

$\mathcal{P}$ = {abbac, ac, bacd, ababc}

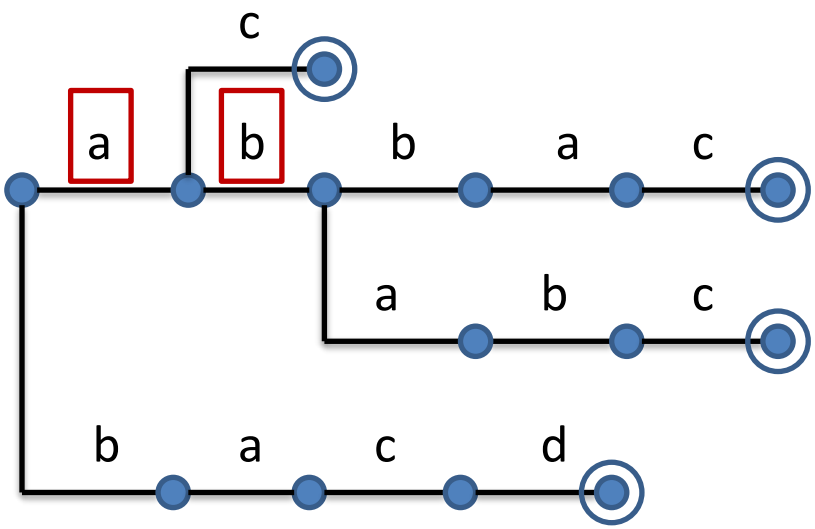


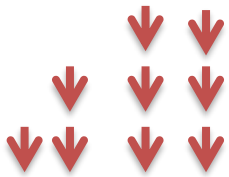


Recherche dans un texte

- Recherche naive: O(mn)

$\mathcal{P}$ = {abbac, ac, bacd, ababc}





$\mathcal{T}$ = a b a b b a c d ….

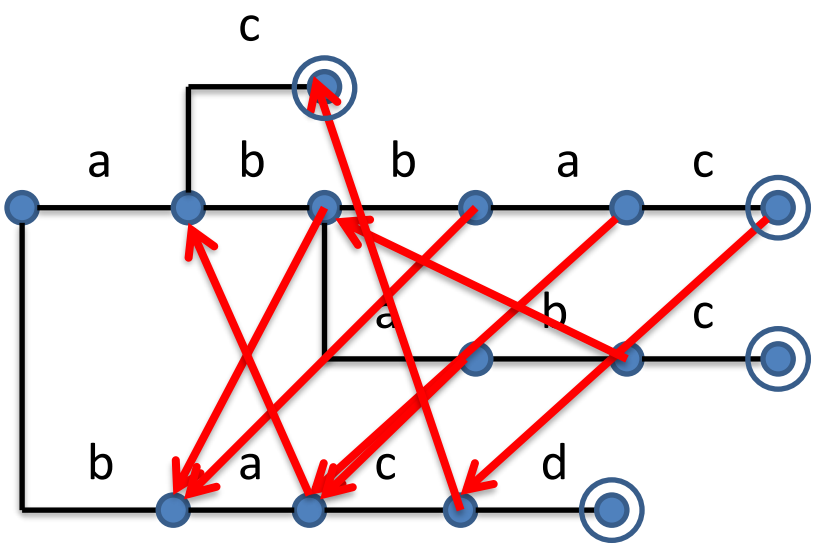
Recherche dans un texte

- Accélération: Généralisation de KMP. Considérer une fonction d'échec.
- *X*: Préfixe d'un mot de $\mathcal{P}$

  Bord(X): Plus long suffixe propre de *X* qui soit préfixe d'un mot de $\mathcal{P}$.
- **Fonction d'échec**: Renvoie au sommet correspondant au plus long bord du mot atteint.

Fonction d'échec

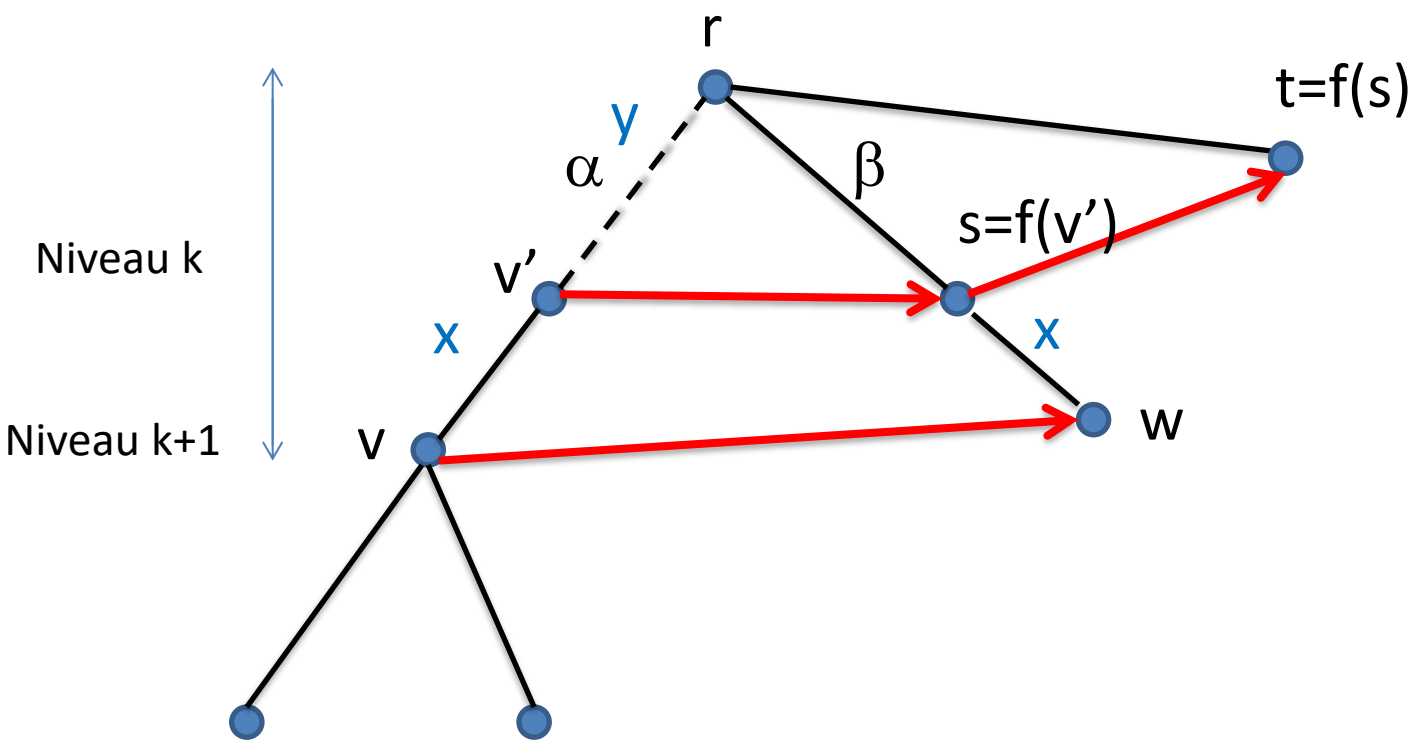
$\mathcal{P}$ = {abbac, ac, bacd, ababc}



Algorithme linéaire pour la fonction d'échec

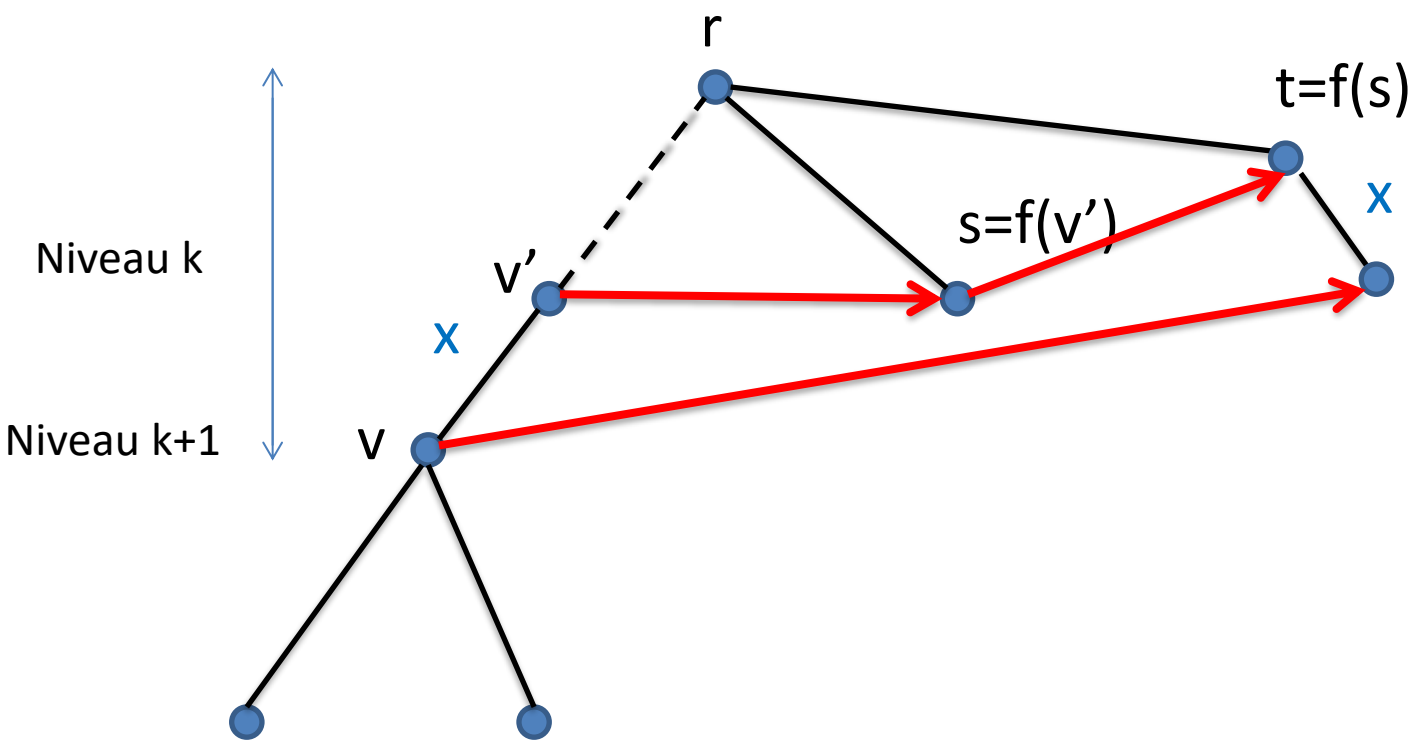
- *r* racine de $\mathcal{K}$
- Calculer *f(v)* pour tout sommet *v* de $\mathcal{K}$.
- *v'* → *v* : arête étiquetée de la lettre *x*
  - Si *v* = *r* ou *r* → *v*, alors *f(v)* = *r*.
  - On suppose que *f(v)* connu pour tout sommet *v* de profondeur ≤ *k*. On veut calculer *f(v)* pour *v* de profondeur *k+1*.

Algorithme linéaire pour la fonction d'échec



- Si il existe une arête *f(v')* → *w* étiquetée *x*, alors *f(v)* = *w*.

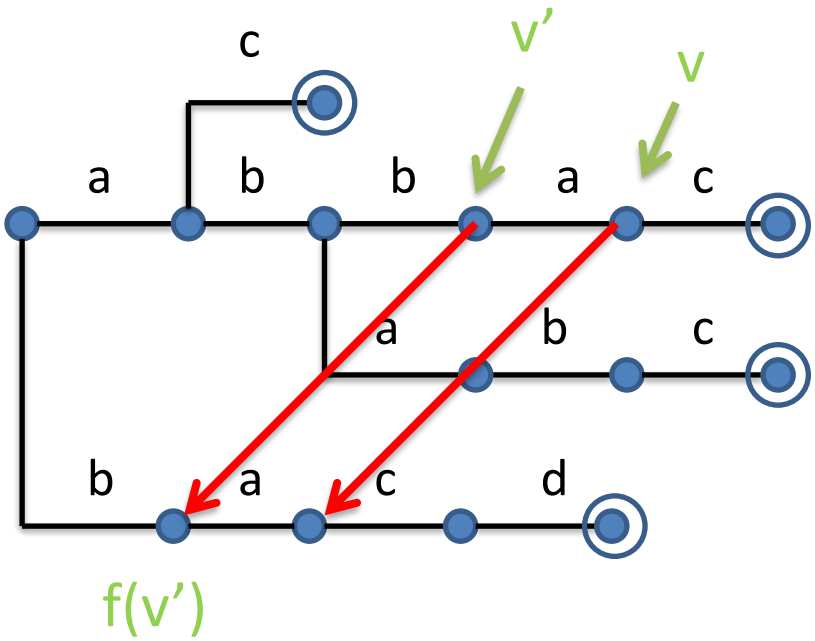
Algorithme linéaire pour la fonction d'échec



- Sinon, on recommence avec s=*f(v')*

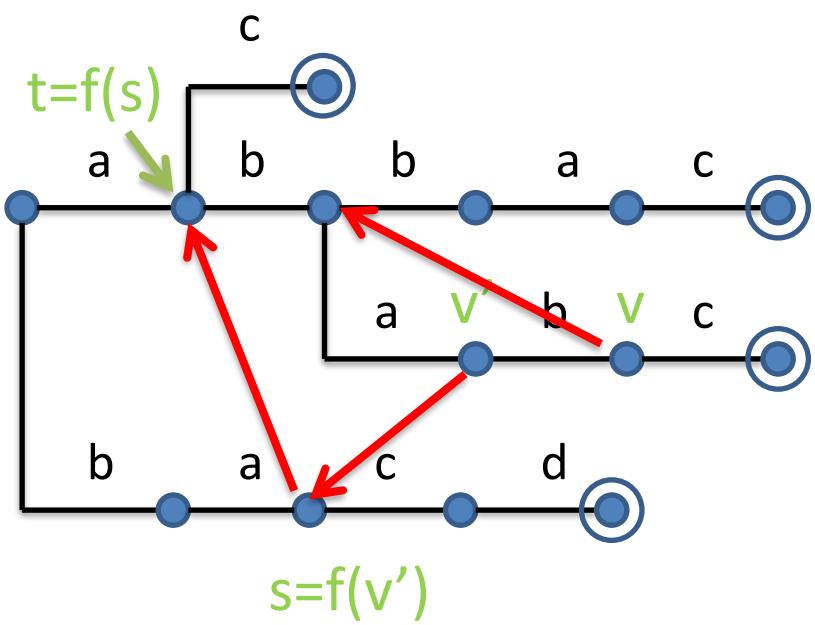
Fonction d'échec

$\mathcal{P}$ = {abbac, ac, bacd, ababc}



Fonction d'échec

$\mathcal{P}$ = {abbac, ac, bacd, ababc}



Algorithme f(v) :

(1) r racine; v' parent de v; x caractère sur l'arc (v, v'); $w = f(v')$;
(2) *Tant que* l'arc qui suit w n'est pas étiquetté x and $w \neq r$ *Faire*
(3) $\quad w = f(w)$;
(4) *Fin Tant que*
(5) *Si* il existe un arc (w, w') sortant de w étiquetté x *alors*
(6) $\quad f(v) = w'$;
(7) *Sinon*
(8) $\quad f(v) = r$;

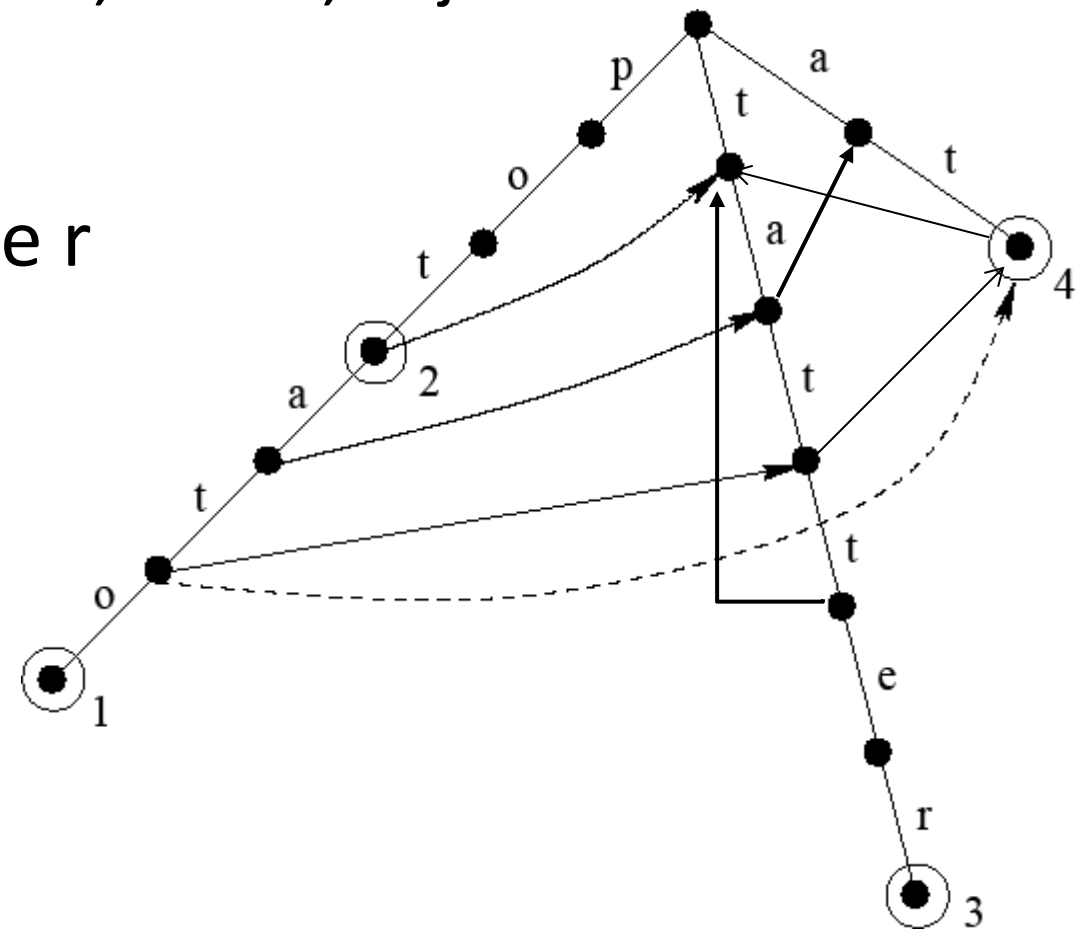
Algorithme AC :

(1) $w :=$ racine;
(2) $j := 1$; {indice sur le texte}
(3) Répéter
(4) $\quad$ Tant que $\delta(w, t_j)$ défini
(5) $\quad\quad w' := \delta(w, t_j)$; $w'' := w'$;
(6) $\quad\quad$ Tant que w'' n'est pas la racine
(7) $\quad\quad\quad$ Si w'' est terminal et étiquetté i
(8) $\quad\quad\quad\quad$ "Occurrence" de P_i à la position j;
(9) $\quad\quad\quad w'' := f(w'')$;
(10) $\quad\quad w := w'$; $j := j + 1$;
(11) $\quad w := f(w)$;
(12) Jusqu'à $j = n$

Exemple

- P = {potato, pot, tatter, at}

T = p o t a t a t t e r



Complexité

- Optimisation: Fonction *op* (pour output)

  Pour tout nœud *v*, C_v chemin, possiblement vide, déterminé par les sommets *f(v), f(f(v)),…* autres que la racine.

  op(v) : Premier nœud terminal de C_v, s'il y a lieu.

  Alors, dans l'algo, remplacer *f* à la ligne (9) par *op*.
- Complexité:
  - Fonctions *f* et *op* calculées en temps *O(m)*
  - *O(n)* comparaisons et *O(k)* parcours des liens *op,* où *k* est le nb d'occurrences des mots de $\mathcal{P}$ dans *T*

  ➔ Parcours de *T* en *O(n+k)*

Références

- *Algorithms on Strings, Trees and Sequences – Computer science and Computational biology,* Dan Gusfield, Cambridge University Press, 1997. Partie I.
- Vous pouvez aussi consulter les cours de Thierry Lecroq sur le sujet, ou les ouvrages de Maxime Crochemore, Mathieu Raffinot, Gonzallo Navarro.