PLAN

* 1. Algorithmes pour la recherche exacte
  * Approche naïve
  * Knuth-Morris-Pratt
  * Boyer-Moore
* 2. Recherche d'un ensemble de mots dans un texte
  * Aho-Corasick

Le problème:

Étant donné un alphabet Σ, un mot $P = p_1 \ldots p_m$ et un texte $T = t_1 \ldots t_n$ où $n >> m$, on veut trouver toutes les positions des occurrences exactes de P dans T

Exemple:

P apparaît dans T aux positions 5,7, 15 et 21.

Intérêts de la recherche exacte:

- Utilitaire de base pour la manipulation de textes.
- Applications courantes:
  - utilitaire UNIX comme grep
  - outils de recherche web
  - recherche dans les catalogues de bibliothèques, les revues électroniques
  - utilitaire de base pour la recherche de motifs biologiques: recherche approchée (BLAST, FASTA), recherche de répétititons, recherche de facteurs de transcription....

Pourquoi a-t-on besoin d'algorithmes rapides pour la recherche exacte?

- Banques de données ou catalogues électroniques gigantesques et croissants de façon exponentielle.

- La recherche exacte est souvent utilisée comme étape de filtrage dans des logiciels complexes -> on veut donc que cette étape se fasse le + rapidement possible

On va voir deux variantes du problème:

- Le mot P est connu à l'avance et peut subir un prétraitement. Ce prétraitement peut se faire, en général en $O(m)$ où m est la longueur du mot P. La recherche dans le texte T, se fait ensuite en $O(n)$, où n est la longueur de T.

Aujourd'hui

- Le texte T est connu à l'avance et peut subir un prétraitement. Le traitement se fait en $O(n)$ et la recherche en $O(m)$.

Arbre des suffixes

La semaine prochaine

1. Algorithmes pour la recherche exacte

a) Approche naïve

On compare P avec chaque sous-mot de T de longueur m

Plus formellement, pour chaque position j dans le texte T, $1 \leq j \leq n - m + 1$, on compare P avec le sous-mot $t_j \ldots t_{j+m-1}$

Si on a un “mismatch”:
- on arrête la comparaison
- on incrémente j ($j \leftarrow j + 1$)
- on recommence la comparaison au début de P

Si on réussit à aligner complètement le mot P:
- on garde en mémoire la position j de l’occurrence
- on incrémente j ($j \leftarrow j + 1$)
- on recommence la comparaison au début de P

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

© notes de cours de Nadia El-Mabrouk

Complexité dans le pire des cas:

Dans le pire des cas, pour chaque position j dans le texte, on compare tout le mot P.

$$\underbrace{m}_{\text{longueur de P}} \times \underbrace{(n-m+1)}_{\substack{\text{nombre de positions dans le texte T} \\ \text{où il est possible d'aligner complètement P}}} \Rightarrow O(nm)$$

Exemple: $P = aaa$

$$T = aaaaaaaaaaaaaaaaaaaa\ldots$$

Idées pour accélérer le calcul:

* Essayer, après un “mismatch”, de faire bouger P de plus d’une position vers la droite en s’assurant de ne pas rater des occurrences du mot P

* À l’étape j+1, ne pas recomparer tous les caractères de P en gardant de l’information sur les caractères comparés à l’étape j

* On va voir maintenant que ces idées vont nous donner des algorithmes de complexité $O(n+m)$

Algorithme de Morris-Pratt (1970)

- Utilise les deux idées discutées précédemment i.e. le déplacement de plus d'une position vers la droite lors d'un mismatch ou lors de la découverte d'une occurrence et ne pas comparer tous les caractères de P après un déplacement.

Exemple:

$$P = \boxed{a\,b\,c\,x\,a\,b\,c}\,d\,e$$
$$T = \ldots \boxed{a\,b\,c\,x\,a\,b\,c}\,e\,\ldots$$

mismatch en t_j

→ Le décalage ne dépend que de P

→ Prétraitement de P pour calculer la valeur des décalages

$$P = a\,b\,c\,x\,a\,b\,c\,d\,e$$
$$T = \ldots \underbrace{a\,b\,c\,x}\,a\,b\,c\,e\,\ldots$$

décalage

on recommence en alignant p_4 et t_j

Prétraitement du mot P en $O(m)$:

- Pour chaque position i dans P, on définit $sp_i(P)$ comme étant la longueur du plus long suffixe propre de $p_1 \ldots p_i$ qui est aussi un préfixe de P

7

P =	a	b	c	x	a	b	c	d	e
suffixe	ε	ε	ε	ε	a	ab	abc	ε	ε
$sp_i(P)$	0	0	0	0	1	2	3	0	0

7

$$P = abcxabcde$$
$$T = \ldots abcxabce \ldots$$

mismatch en t_j

$$P = abcxabcde$$
$$T = \ldots abcxabce \ldots$$

on recommence en alignant p_4 et t_j → $sp_7(P) + 1$

Algorithme:

Algorithme Morris-Prath

occurrences={}

$i \leftarrow 1$

$j \leftarrow 1$

TANT QUE $j \leq n$ FAIRE

SI $P_i \neq T_j$ ALORS

$j \leftarrow j + 1$

$i \leftarrow 1$

SINON

TANT QUE $P_i = T_j$ et $i \leq m$ FAIRE

$i \leftarrow i + 1$

$j \leftarrow j + 1$

FIN TANT QUE

SI $i = m + 1$ ALORS

occurrences $\leftarrow$ occurrences $\cup \{j - m\}$

$i \leftarrow sp_{i-1}(P) + 1$

SINON

$i \leftarrow sp_{i-1}(P) + 1$

FIN SI

FIN SI

FIN TANT QUE

RETOURNER occurrences

Algorithme de Knuth-Morris-Pratt (1977)

D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. SIAM J. Computing, 6(2):323--350, 1977.

- Version optimisée de l'algorithme de Morris-Pratt

- L'optimisation se fait dans le prétraitement du mot P

Prétraitement du mot P (version améliorée):

- Pour chaque position i dans P, on définit $sp_i'(P)$ comme étant la longueur du plus long suffixe propre de $p_1 \cdots p_i$ qui est aussi un préfixe de P, avec la condition supplémentaire que $p_{i+1} \neq p_{sp_i+1}$

P =	a	b	c	x	a	b	c	d	e
suffix	ε	ε	ε	ε	a	ab	abc	ε	ε
$sp_i(P)$	0	0	0	0	1	2	3	0	0
$sp_i'(P)$	0	0	0	0	0	0	3	0	0

Pourquoi: $T = \ldots abcxaabd \ldots$

$P = abcxabcde$

mismatch en t_j

avec $sp_i(P)$:

$T = \ldots abcxaabd \ldots$ (t_j)

$P = abcxabcde$

avec $sp_i'(P)$:

$T = \ldots abcxaabd \ldots$ (t_j)

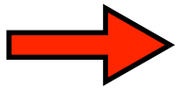
$P = abcxabcde$

Complexité: - prétraitement du mot P -> $O(m)$

- un passage de T -> $O(n)$

- complexité totale -> $O(n+m)$

À voir: Est-ce qu'on peut manquer des occurrences de P avec la règle de décalage de Knuth-Morris-Prath i.e comment prouver que l'algorithme trouve bien toutes les occurrences de P dans T??



Réponse au tableau!!

Algorithme de Boyer-Moore *

- Comme pour l'approche naïve on compare P avec T et lorsqu'on a un "mismatch" ou une occurrence de P, on déplace P vers la droite.

Différences avec l'approche naïve:

- On compare P avec T de droite à gauche (de p_m à p_1)
- Deux règles de déplacement:
  1) règle du "mauvais caractère"
  2) règle du "bon suffixe"
- La complexité dans le pire des cas est de $O(nm)$
- Par contre, quelques modifications simples par Galil*, donne un algorithme dont la complexité dans le pire cas est $O(n+m)$

* R.S. Boyer and J.S.Moore, *A fast string searching algorithm*, Comm. ACM, 20, pp. 762-772, 1977.

* Z. Galil, *On improving the worst case running time of the Boyer-Moore algorithm*, Comm. ACM, 22, pp. 505-508, 1979.

La règle du mauvais caractère:

$$T = \quad \ldots p \ldots$$
$$P = t\,p\,a\,b\,x\,a\,b$$

mismatch

Si on connaît le “p” le plus à droite dans le mot P alors on peut déplacer le mot P pour que ce “p” le plus à droite soit sous le “p” dans le texte

$$T = \quad \ldots p \ldots$$
$$P = t\,p\,a\,b\,x\,a\,b$$

match on recommence à comparer ici

Tout déplacement plus petit entraîne un rejet immédiat.

Prétraitement du mot P en $O(m)$:

- Pour chaque $x \in \Sigma$, on définit $R(x)$ comme étant la position de l'occurrence de x la plus à droite dans le mot P. Si x n'apparaît pas dans P alors $R(x) = 0$

$$\begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ P = & t & p & a & b & x & a & b \end{matrix}$$

$$R(b) = 7$$
$$R(a) = 6$$
$$R(x) = 5$$
$$R(p) = 2$$
$$R(t) = 1$$

Algorithme Calcul R
  alphabet $\leftarrow \Sigma$
  $i \leftarrow m$ (la longueur de P)
  TANT QUE $i \neq 0$ FAIRE
    SI $p_i \in$ alphabet ALORS
      $R(p_i) = i$
      alphabet $\leftarrow$ alphabet$\backslash\{p_i\}$
      $i \leftarrow i - 1$
    SINON
      $i \leftarrow i - 1$
    FIN SI
  FIN TANT QUE
  SI alphabet$\neq \{\}$ ALORS
    $\forall x \in$ alphabet, $R(x) = 0$
  FIN SI
  RETOURNER R

$$\begin{matrix} & & & t_k \\ & & & \Downarrow \\ T = & & \ldots & p & \ldots \end{matrix}$$
$$P = t\ p\ \underbrace{a\ b\ x\ a\ b}$$

décalage $\max\{i - R(T_k), 1\}$

La règle du bon suffixe:

$$T = \ldots \quad x\boxed{\,t\,}\ldots$$
$$P = \ldots \quad y\boxed{\,t\,}$$

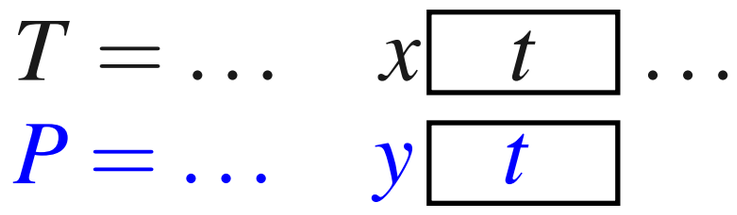
1er cas: P contient au moins une autre occurrence du mot t

- on trouve l'occurrence de t la plus à droite qui n'est pas précédée d'un "y"
- on déplace cette occurrence pour quelle soit située sous celle dans le texte

$$T = \ldots \quad x\boxed{\,t\,}\ldots$$
$$P = \ldots \quad z\boxed{\,t\,} \quad y\boxed{\,t\,}$$

$$T = \ldots \quad x\boxed{\,t\,}\ldots$$
$$P = \ldots \quad z\boxed{\,t\,} \quad y\boxed{\,t\,}$$

La règle du bon suffixe (suite):



2ème cas: P ne contient pas d'autre occurrence du mot t

- on utilise la règle de décalage de Knuth-Morris-Prath

L'algorithme:

- On commence par aligner P avec le préfixe de taille m de T (comparaison de droite à gauche)
- Si on trouve un "mismatch" ou une occurrence, on décale P du maximum entre les valeurs de décalage obtenues par la règle du mauvais caractère et la règle du bon suffixe et on recommence la comparaison de droite à gauche
- On arrête lorsqu'on a aligné P avec le suffixe de taille m de T ou lorsqu'un déplacement envoie le dernier caractère de P plus loin que la fin de T

Prétraitement ”bon suffixe”:

Le problème:

On veut calculer pour chaque position i d’un mot $P = p_1 \ldots p_m$ $i > 1$, $Z_i(P)$, la longueur du + long sous-mot $SM_i(P)$ commençant en i qui est identique à un préfixe de P

Approche naïve:

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

Comparer P , caractère par caractère avec P[i..m], pour tout i

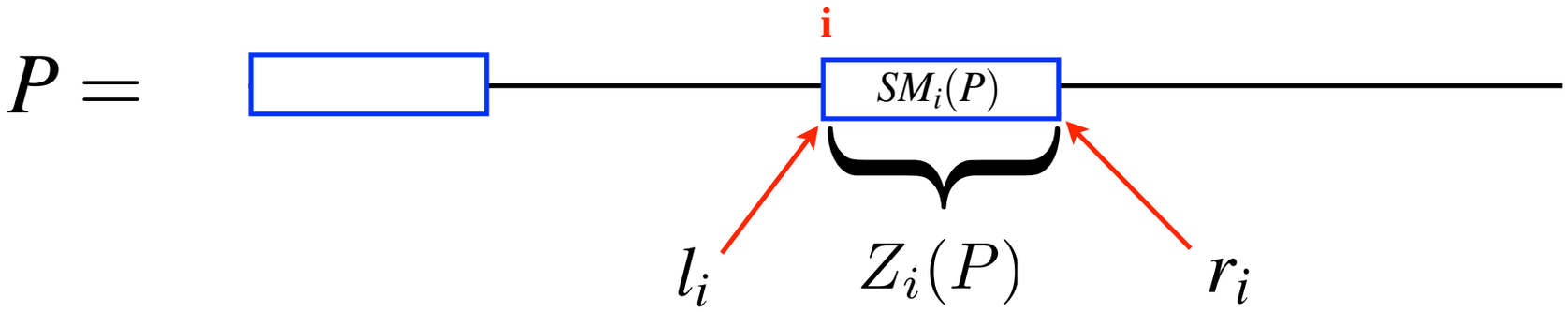
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
P[2..m] =	A	C	A	A	C	G	A	T	A	A	C	A	A	C	G	G

$$SM_2(P) = A \longrightarrow Z_2(P) = 1$$

Complexité en temps: $O(m^2)$

Idée:

Utiliser les $Z_i(P)$ déjà calculés pour calculer les $Z_k(P), k > i$



On va garder en mémoire tous les $Z_i(P)$ déjà calculés et aussi

$r = r_i$ le plus à droite déjà calculé

$l =$ la position à gauche qui lui est associée

L'algorithme:

- On calcule $Z_2(P)$

- Étant donné les $Z_i(P), 2 \leq i \leq k-1$, et les valeurs courantes de droite r et gauche l, on calcule $Z_k(P)$ comme suit:

1) Si k > r, trouver $Z_k(P)$ en comparant les caractères de P et P[k..m] deux à deux jusqu'à un "mismatch"

Si $Z_k(P) > 0$ alors

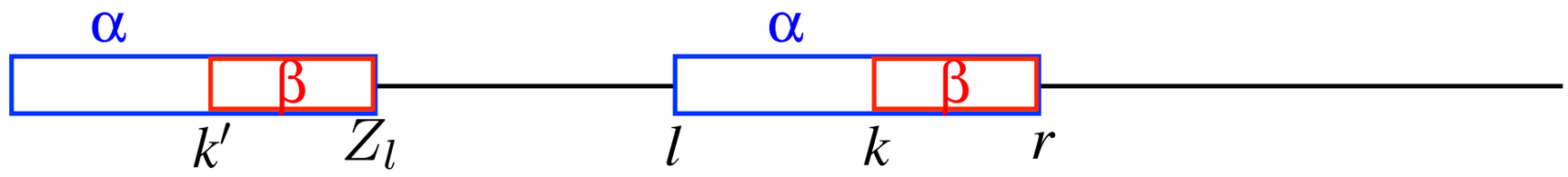
r ⟵ k + $Z_k(P)$ - 1

l ⟵ k

(Sinon pas de changement)

L'algorithme (suite):

2) Si $k \leq r$:

$P =$ 

Le sous-mot (ou une partie du sous-mot) ß commençant en position k doit être identique à un préfixe de P de longueur au moins égale au minimum entre |ß| et $Z_{k'}(P)$

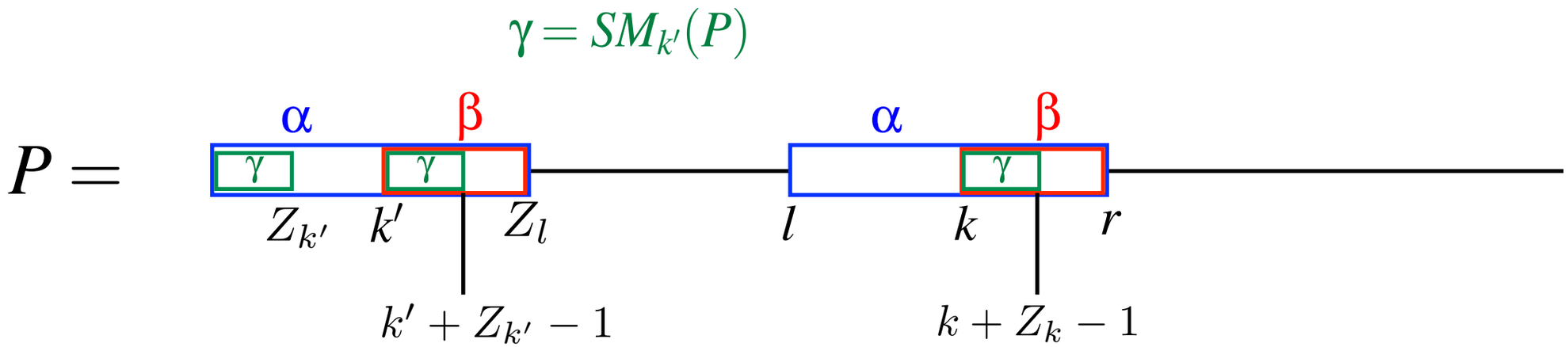
On va séparer 2) en 2 cas distincts:

2a) $Z_{k'}(P) <$ |ß|

2b) $Z_{k'}(P) \geq$ |ß|

L'algorithme (suite):

2a) $Z_{k'}(P) <$ lßl



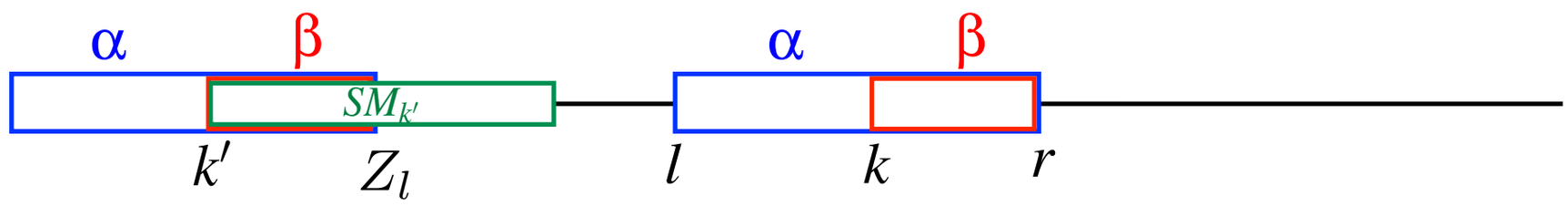
Alors:

- $Z_k(P) = Z_{k'}(P)$

- r et l demeurent inchangés

L'algorithme (suite):

2b) $Z_{k'}(P) \geq$ lßl

$P =$ 

- le sous-mot P[k..r] est un préfixe de P
- On compare les caractères deux à deux en commençant avec les positions r+1 et lαl + 1, jusqu'à un "mismatch" disons en position q > r

$$Z_k \leftarrow q - k$$
$$r \leftarrow q - 1$$
$$l \leftarrow k$$

2. Recherche d'un ensemble de mots dans un texte

Le problème: Étant donné un alphabet Σ, un ensemble de mots $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ et un texte $T = t_1 \ldots t_n$, on veut trouver toutes les occurrences exactes de tous les mots de $\mathcal{P}$ dans T. Les occurrences peuvent se chevaucher.

a) Approche naïve

Utiliser l'algorithme de Boyer-Moore ou Knuth-Morris-Prath et comparer chacun des mots P_i avec le texte T.

Complexité: $O(kn + M)$ où $M = \sum_{i=1}^{k} longueur(P_i)$

Algorithme de Aho-Corasick*

- Permet de trouver toutes les occurrences de tous les mots en un seul passage dans le texte T

- Complexité: $O(n+M+k)$ où M est la somme des longueurs des mots cherchés et k, le nombre total d'occurrences.

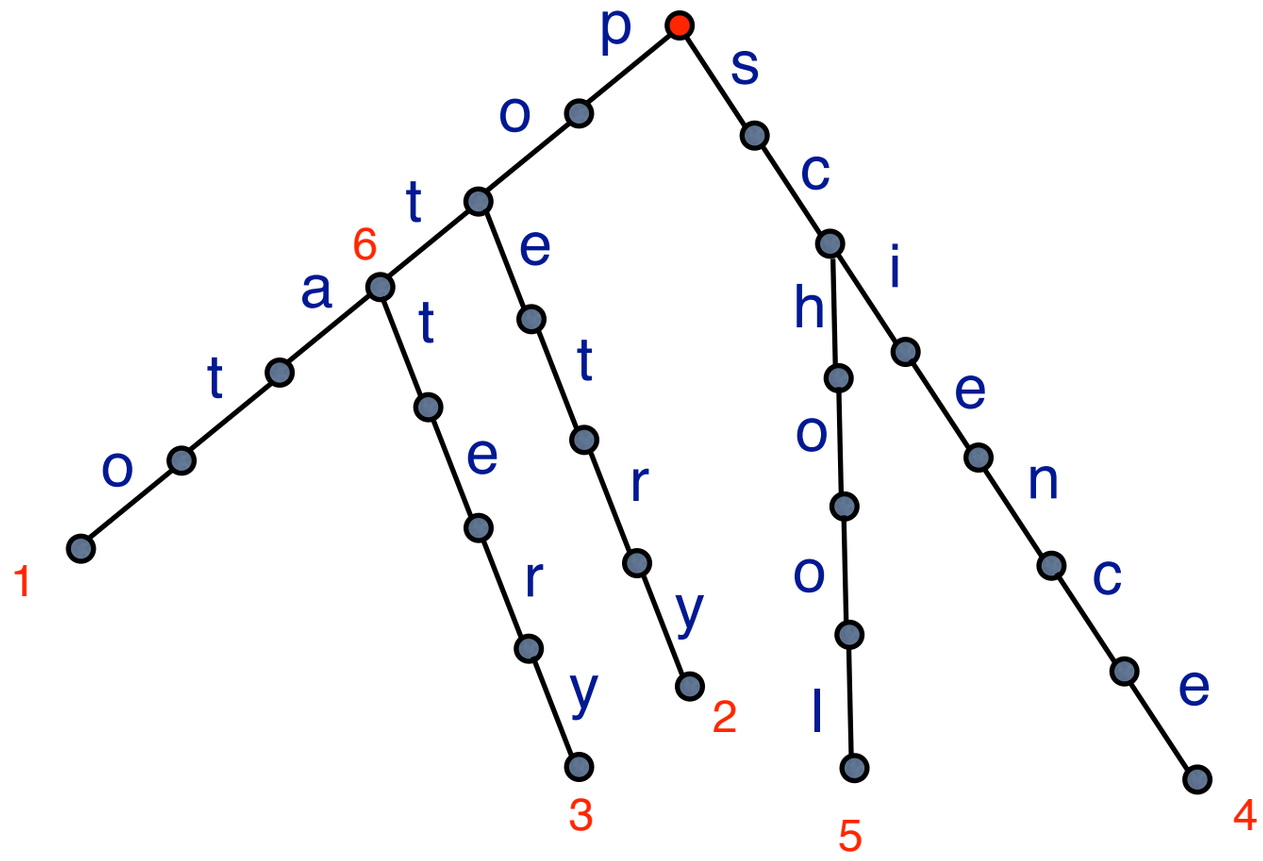
- L'algorithme est basé sur la construction d'un arbre $\mathcal{K}$ à partir des mots de l'ensemble $\mathcal{P}$

- L'arbre $\mathcal{K}$ est constitué d'un ensemble de noeuds, terminaux ou non, et de transitions
  - Chaque transition est étiquettée par exactement un caractère
  - Deux transitions différentes à partir d'un même noeud ont des étiquettes différentes
  - L'étiquette du chemin de la racine à un noeud v non terminal correspond à un préfixe propre d'un mot de $\mathcal{P}$. Chaque noeud terminal correspond à un mot de $\mathcal{P}$

* Aho, A. et Corasick, M., Efficient string matching: an aid to bibliographic search, Comm. ACM, 18, pp.333-340, 1975.

Construction de l'arbre $\mathcal{K}$ en O(M):

$\mathcal{P}$ = {potato, poetry, pottery, science, school, pot}



Utilisation naïve de l'arbre: Pour chaque position j du texte T, suivre l'unique chemin jusqu'à une feuille ou un "mismatch" et récupérer les étiquettes des occurrences visitées

Complexité: $O(nM)$

Idée de Aho-Corasick: Utiliser l'idée de Knuth-Morris-Prath conjointement avec l'arbre $\mathcal{K}$

Assomption: Aucun mot de $\mathcal{P}$ est un sous-mot non-préfixe d'un autre mot de $\mathcal{P}$

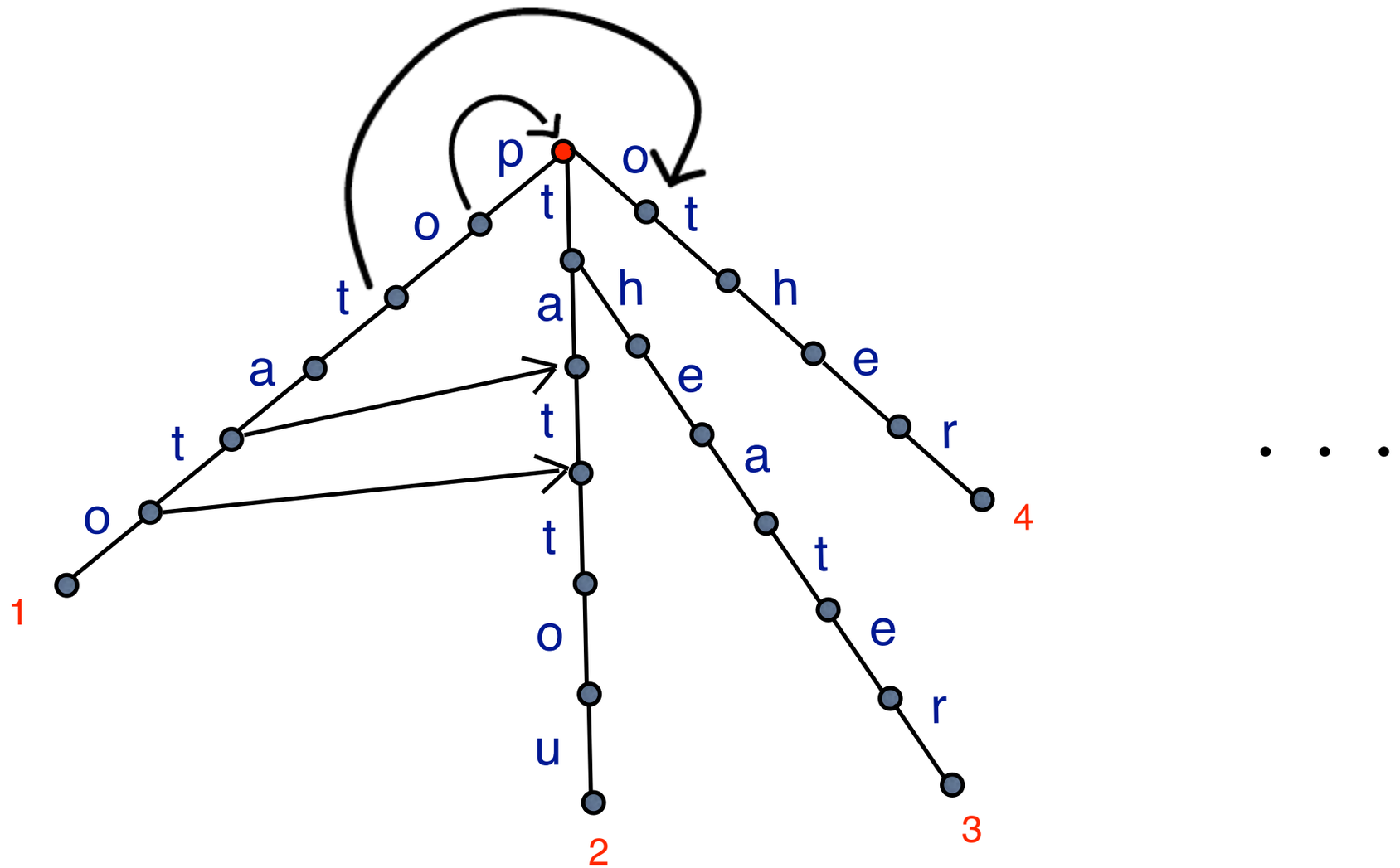
- Pour chaque noeud v dans $\mathcal{K}$, on définit L(v) comme étant l'étiquette du chemin de la racine au noeud v.

- Soit n_v l'unique noeud tel que $L(n_v)$ est le plus long suffixe propre de L(v) qui est aussi un préfixe d'un mot de $\mathcal{P}$ (si le plus long suffixe est le mot vide alors n_v est la racine de l'arbre)

- Pour chaque noeud v de l'arbre ajouter un lien (failure link) de v à n_v

Exemple:

$\mathcal{P}$ = {potato, tattou, theater, other}
1 2 3 4



. . .

Algorithme de Aho-Corasick:

Algorithme Aho-Corasick

occurrences={}

$l \leftarrow 1$ position où l'on commence l'alignement

$c \leftarrow 1$ position courante dans T

$w \leftarrow$ racine de l'arbre

TANT QUE $c < n$ FAIRE

TANT QU'il y a une arete (w, w') d'etiquette $T(c)$ FAIRE

SI w' porte une etiquette i ALORS

occurrences $\leftarrow$ occurrences $\cup \{(P_i, l)\}$

FIN SI

$w \leftarrow w'$

$c \leftarrow c + 1$

FIN TANT QUE

$w \leftarrow n_w$

$l \leftarrow c - |L(w)|$

FIN TANT QUE

RETOURNER occurrences