Algorithmique et Alignement de Chaînes
Recherche de Motifs - Algorithmes à base d'index

G. Fertin
guillaume.fertin@univ-nantes.fr

Université de Nantes, LS2N
Bât 34 — Bureau 301

M2 ATAL — 2020/2021

Sommaire

Recherche de Motif (=Pattern Matching)

Instance :

- un motif P de longueur m
- un texte T de longueur n

Question : à quelle(s) position(s) P apparaît-il dans T ?

Hypothèses de travail :

- $m << n$, et on supposera n très grand
- le motif et le texte sont sur un alphabet Σ de taille σ
- doit-on considérer que $\sigma = O(1)$?
  $\rightarrow$ on discutera les deux possibilités

Recherche de Motif - Applications

- recherche d'un mot/une locution dans un texte :
  - ex : grep, traitement de texte, boîte mail, etc. (Ctrl-F)
  - $\sigma \leq 100$ (26 minuscules, 26 majuscules, caractères spéciaux), $n \in [10^3; 10^6]$

- bio-informatique : motifs biologiquement significatifs dans des séquences
  - ADN : $\sigma = 4$, $n \in [10^3; 10^9]$
  - Protéines : $\sigma = 20$, $n \in [10^3; 10^5]$
  - Gènes : $\sigma \in [10^3; 10^4]$, $n \in [10^3; 10^5] \rightarrow$ peu de lettres répétées

Quelques notations/définitions

- mot = suite ordonnée de lettres de Σ
- mot vide : noté ε

- un mot u est préfixe d'un mot w s'il existe un mot v (éventuellement vide) tel que

$$w = uv$$

- un mot v est suffixe d'un mot w s'il existe un mot u (éventuellement vide) tel que

$$w = uv$$

Paramètres observés

Soit A un algorithme résolvant le pattern matching

Deux paramètres observés :

- espace mémoire requis par A
- temps d'exécution de A

Paramètres observés

Espace mémoire :

- stocker $T \rightarrow O(n \log \sigma)$ ($O(n)$ si on considère $\sigma = O(1)$)
- stocker $P \rightarrow O(m \log \sigma)$ ($O(m)$ si on considère $\sigma = O(1)$)
- $\Rightarrow$ but : s'approcher le plus possible de $O((n+m) \log \sigma)$ (donc de $O(n \log \sigma)$ puisque $m \leq n$)

Paramètres observés

Temps d'exécution :

- rappel : n très grand
- $\Rightarrow$ pas d'algorithme (par exemple) en $O(n^2)$ ni $O(nm)$!
- $\Rightarrow$ but : complexité linéaire, càd en $O(n+m)$ ($= O(n)$ puisque $m \leq n$)
- Remarque : si P apparaît occ fois dans T :
  - possibilité d'intégrer occ à la complexité
  - ex : $O(n + occ)$
  - $\rightarrow$ description plus fine de la complexité

Algorithmes à base d'index

- pré-traitement des données d'entrée (P ou T)
- création d'une structure (d'index) suite à ce pré-traitement

Ceci impacte sur :

- l'espace mémoire requis (stocker l'index)
- le temps d'exécution (construire l'index) $\rightarrow$ amorti si requêtes multiples

Paramètres observés (bis)

Classiquement, on sépare les temps de construction de l'index et de recherche de motif $\rightarrow$ 3 paramètres :

- Temps de construction de l'index
- Temps de la recherche de motifs
- Espace requis (index inclus)

Les 3 index vus dans ce cours

1. Automate (pré-traitement de P)
2. Arbre des suffixes (pré-traitement de T)
3. Tableau des suffixes (pré-traitement de T)
4. FM-index (pré-traitement de T) — sur un autre jeu de transparents

Sommaire

Algorithme à base d'automate

- on suppose que P est connu à l'avance (hypothèse raisonnable)
- T peut ne pas l'être

$\Rightarrow$ pré-traitement sur P

Algorithme à base d'automate

$\Rightarrow$ construction de l'Automate Fini Déterministe (AFD) $\mathcal{A}(P)$ qui reconnaît le langage $\Sigma^* P$

$\mathcal{A}(P)$ reconnaît le langage $\Sigma^* P \Leftrightarrow$ tout mot se terminant par P sera reconnu par l'automate

$\Rightarrow$ la lecture par l'AFD de tout mot se terminant par P aboutira sur l'état final

AFD qui reconnaît $\Sigma^* P$

Un automate $\mathcal{A} = (Q, q_0, \mathcal{T}, E)$ se définit par :

- Q = l'ensemble des états de l'automate
- q_0 = état initial
- $\mathcal{T}$ = ensemble des états finaux (ici, un seul)
- E = ensemble des transitions (passage d'un état à l'autre)
  Ici, chaque transition sera étiquetée par une lettre de Σ

Définition de l'AFD

- $Q =$ ensemble des préfixes de P (mot vide ε et P inclus)
- $q_0 = \varepsilon$
- $\mathcal{T} = P$
- pour tout mot $q \in Q$ (q est donc un préfixe de P) et pour toute lettre $a \in \Sigma$,
  - $(q, a, qa) \in E$ ssi qa est aussi un préfixe de P
  - sinon $(q, a, q') \in E$ lorsque q' est le plus long suffixe de qa qui est aussi un préfixe de P

Exemple : Construction de l'AFD pour le motif $P =$ `GCAGAGAG`

Algorithme de recherche

Principe de l'algorithme de Recherche de Motif utilisant l'AFD :

- avancer dans T (en démarrant à $T[1]$, la première lettre de T) en suivant les transitions dans l'automate
- à chaque fois qu'on arrive sur l'état final en ayant lu $T[i]$, indiquer qu'une occurrence de P apparaît en $T[i - m + 1]$

Exemple : Recherche de P dans le texte $T =$ `GCATCGCAGAGAGTATACAGTACG`

Remarque : sur l'exemple, on a effectué 24 comparaisons de caractères (c'est tout simplement n, la longueur du texte T)

Analyse de la méthode

Temps de Construction de $\mathcal{A}(P)$

- Nombre d'états : $m + 1$
- Nombre de transitions : $(m + 1)\sigma$
- Construction de l'automate : pour chaque état q et chaque transition a, il faut calculer l'état dans lequel on aboutit en suivant a partant de q
- On admettra que ce temps de calcul, pour toutes les transitions possibles, est en $O(m\sigma)$ (on ne le démontrera pas)

Analyse de la méthode

Temps de Recherche de Motif

- supposons que $\mathcal{A}(P)$ est codé par une matrice à $m+1$ lignes (états) et σ colonnes (transitions)

Exemple : de codage des transitions sur l'AFD exemple

État	Préfixe de P	A	C	G	T
0	ε	0	0	1	0
1	G	0	2	0	0
2	GC	3	0	1	0
3	GCA	0	0	4	0
4	GCAG	5	2	1	0
5	GCAGA	0	0	6	0
6	GCAGAG	7	2	1	0
7	GCAGAGA	0	0	8	0
8	GCAGAGAG	0	2	1	0

Analyse de la méthode

Temps de Recherche de Motif

- permet de passer d'un état à l'autre (dans l'algorithme de recherche) en temps constant $O(1)$
- $\rightarrow$ chaque lecture d'un caractère de T (et déplacement dans $\mathcal{A}(P)$) coûte $O(1)$
- Au total : $O(n)$

Espace Requis :
Matrice codant $\mathcal{A}(P)$ ($+$ état initial, état final) $\Rightarrow$ $O(m\sigma)$

En résumé

	Construction Index	Recherche de Motifs	Espace Requis (Index)
σ non borné	$O(m\sigma)$	$O(n)$	$O(m\sigma)$
σ borné	$O(m)$	$O(n)$	$O(m)$

En résumé (suite et fin)

Si on considère que $\sigma = O(1)$, alors :

- complexité totale en temps linéaire, en $O(n + m)$
- espace mémoire total requis linéaire $O(n + m)$
- $\rightarrow$ temps optimal si σ constant... mais finalement peu utilisé (KMP ou Boyer-Moore généralement préférés)
- si σ non constant : problème
  - il existe des codages plus complexes pour stocker $\mathcal{A}(P) \rightarrow$ place mémoire diminuée
  - ...au prix d'un coût en temps plus élevé

Sommaire

Idée principale

Pré-traiter non pas le motif P, mais le texte T

$\rightarrow$ possible quand le texte T est connu à l'avance, donc *statique*

Exemple : applications biologiques (séquence d'ADN, de protéines, de gènes)

Arbre des Suffixes – Définition

Arbre des suffixes $AS(T)$: arbre qui contient *tous les suffixes* d'un texte T
Plus précisément :

- chaque arête de l'arbre : étiquetée par un caractère de T
- deux arêtes issues d'un même nœud portent des caractères différents
- appelons r la racine de $AS(T)$ et f une de ses feuilles
  appelons $r \rightarrow f$ le chemin dans $AS(T)$ de r vers f
  $\Rightarrow$ les caractères lus le long de $r \rightarrow f$ forment un *suffixe s de T*
- feuilles numérotées de 1 à n (numéro = position du premier caractère de s dans T)

Exemple : Arbre des suffixes de $T =$ `ATAGT`

Caractère de terminaison

Problème 1 : l'arbre des suffixes de $T = \texttt{ATAGT}$ possède 5 suffixes mais seulement 4 feuilles !

Solution :

- ajout (artificiel) d'un caractère de terminaison à T
- ce caractère doit être unique (donc n'apparaît pas dans T)
- par convention, on utilise le caractère \$

Exemple : Arbre des suffixes de $T = \texttt{ATAGT\$}$

Analyse express

Problème 2 : espace mémoire en $O(n^2)$!
En effet :

- n suffixes (longueurs 1 à n)
- potentiellement jusqu'à $1 + 2 + \ldots n = \frac{n(n+1)}{2}$ arêtes dans $AS(T)$

$\rightarrow$ pour la plupart des applications (typiquement, $n > 1000$), inapplicable !

$\Rightarrow$ utiliser une représentation compacte

Représentation Compacte

- tout chemin "sans fourche" dans l'arbre est "compressé" en une *unique arête*
- étiquette de cette arête = concaténation des caractères trouvés sur ce chemin

ATTENTION à coder cette étiquette de façon intelligente !

- si codage = sous-séquence de T, on n'a rien gagné
- à la place, codage = intervalle $[i, j]$, tel que $T[i..j]$ = l'étiquette

Exemple : Construction de l'arbre des suffixes de $T =$ `ATAGT$` sous forme compacte

Représentation Compacte

Propriété : l'arbre des suffixes sous forme compacte possède un nombre de nœuds (et donc d'arêtes) en $O(n)$

Preuve : imaginons construire un $AS(T)$ compact, en insérant (partant d'un arbre vide) tous les suffixes de longueur $n+1-i$, pour tout i allant de 1 à n (= du plus long au plus court)

- pour chaque $2 \leq i \leq n$, insertion du suffixe de longueur $n+1-i \rightarrow$ on crée une fourche
- chaque création de fourche crée **au plus un** nœud interne
- $\Rightarrow$ on a donc au plus $n-1$ nœuds internes

- $\Rightarrow$ au total, au plus $2n-1$ nœuds dans l'arbre ($n-1$ nœuds internes + n feuilles)
- $\Rightarrow$ $O(n)$ nœuds — et donc $O(n)$ arêtes puisque c'est un arbre

Espace mémoire requis pour $AS(T)$

- arbre à $O(n)$ nœuds (et donc $O(n)$ arêtes)
  $\Rightarrow$ structure d'arbre codable en $O(n)$

- informations sur les feuilles : entier de 1 à n
  - $\lceil \log_2 n \rceil$ bits par feuille
  - il y a n feuilles

  $\Rightarrow$ informations sur les feuilles codables en $O(n \log n)$

- Informations sur les arêtes : deux entiers i, j tels que $1 \leq i \leq j \leq n$
  - $2\lceil \log_2 n \rceil$ bits par arête
  - il y a $O(n)$ arêtes

  $\Rightarrow$ informations sur les arêtes codables en $O(n \log n)$

Au total : espace mémoire requis pour $AS(T)$ en $O(n \log n)$

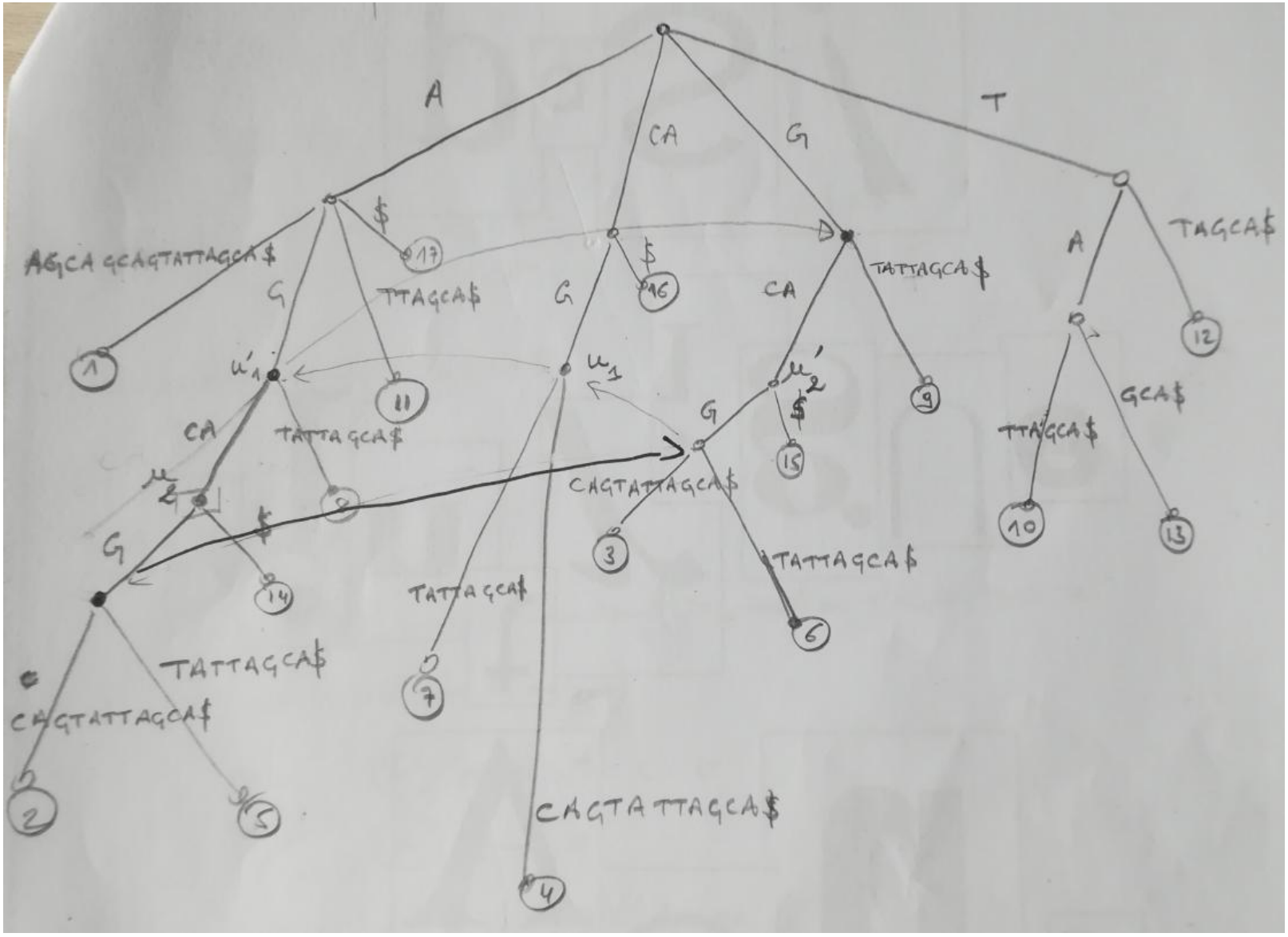
Recherche de motifs dans $AS(T)$

Rappel : motif P, texte T

1. construction de l'arbre des suffixes de T, $AS(T)$
2. partant de la racine, suivre les caractères de P le long d'un unique chemin dans $AS(T)$ jusqu'à ce que :
   - soit une différence apparaît $\rightarrow$ P n'apparaît pas dans T
   - soit on va au bout du motif P :
     - Attention ! Cela peut arriver "au milieu" d'une arête (pas gênant)
     - dans tous les cas, parcours du sous-arbre de $AS(T)$ dans lequel on se trouve : les numéros des feuilles donnent toutes les positions d'apparition de P dans T

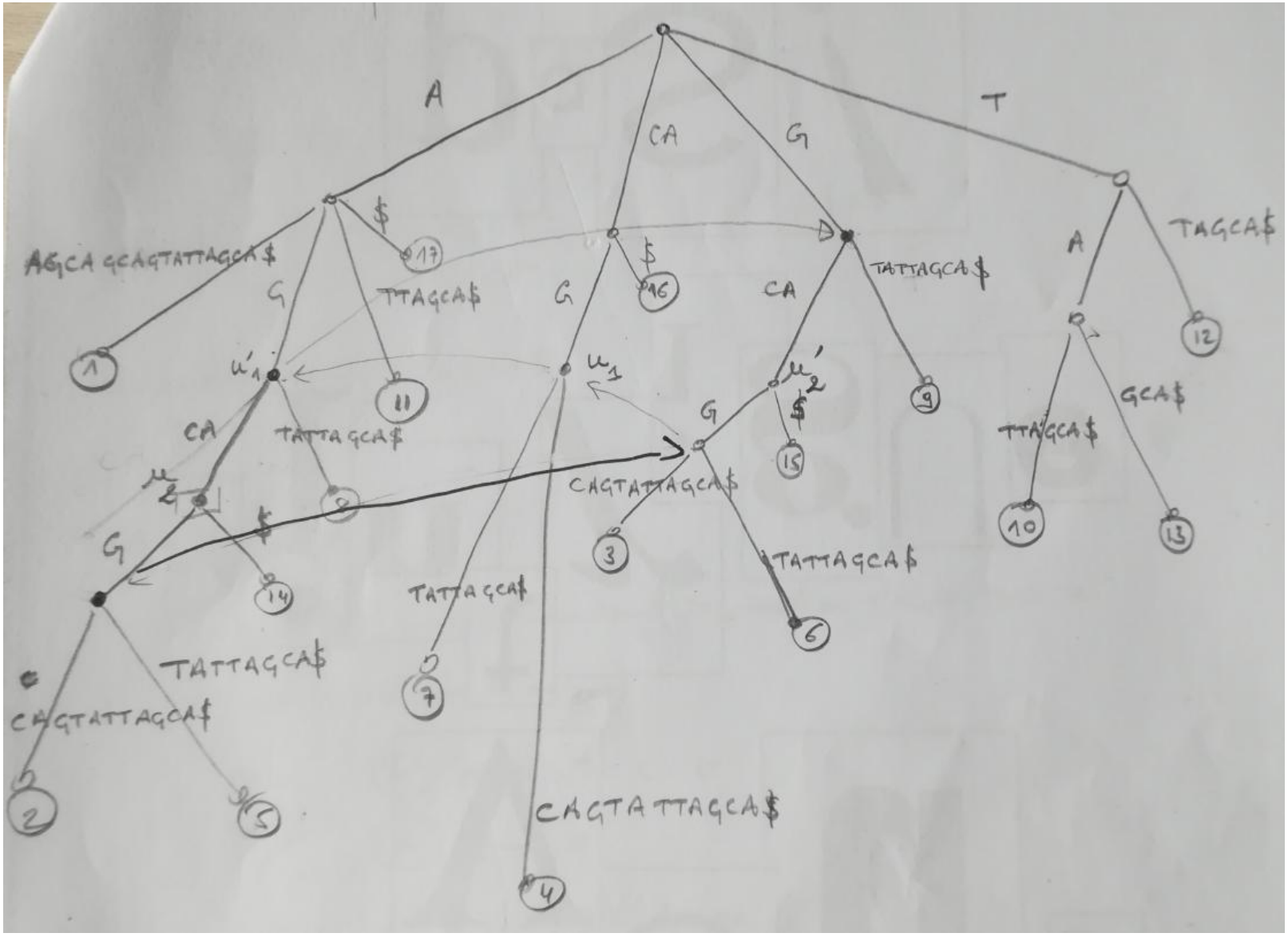
Recherche de motifs dans $AS(T)$

Exemple 1 : recherche de $P =$ AGCT dans $T =$ AAGCAGCAGTATTAGCA



Recherche de motifs dans $AS(T)$

Exemple 2 : recherche de $P =$ AGCA dans $T =$ AAGCAGCAGTATTAGCA



Complexité de la Recherche de Motifs

- descente dans l'arbre au niveau $m \rightarrow O(m\sigma)$: au pire, σ voisins à consulter à partir d'un nœud
- si succès, parcours du sous-arbre (occ feuilles) $\rightarrow O(occ)$

$\Rightarrow$ Complexité en temps de la Recherche de Motif : $O(m\sigma + occ)$
(si $\sigma = O(1)$, alors $O(m + occ)$)

Construction de l'Arbre des Suffixes

Faisons un point :

Construction Index	Recherche de Motifs	Espace Requis (Index)
???	$O(m\sigma + occ)$	$O(n \log n)$

- [+] : Recherche de motifs rapide $O(m\sigma + occ)$... au prix de
- [−] : stockage $O(n \log n)$ pour $AS(T)$ (au lieu de $O(n \log \sigma)$ pour T)
- Reste le temps de construction $\rightarrow$ il ne doit pas dépasser la taille de $AS(T)$, sinon aucun intérêt !

Algorithme simple... mais trop coûteux

Algorithme incrémental : on fait grandir l'arbre

- T_i = l'arbre qui encode tous les suffixes de longueurs n à $n - i + 1$ $(1 \leq i \leq n)$
- T_1 = unique branche d'étiquette $T\$$
- On construit T_{i+1} à partir de T_i :
  - en suivant autant que possible une branche existante
  - en "cassant" cette branche (i.e. en créant une fourche) lorsqu'une différence apparaît
- Complexité : $O(n^2)$ pour un texte T de longueur n

Algorithmes de Construction Linéaire

Il en existe plusieurs :

- Weiner (1973)
- McCreight (1976)
- Ukkonen (1995)
- Farach (1997))

$\rightarrow$ tous en $O(n)$

Différences de conception entre ces algorithmes : compromis entre "algo compréhensible" et "algo implémentable"

Le plus utilisé en pratique est Ukkonen(1995)

Algorithmes de Construction Linéaire

Dans tous les cas :

- construction linéaire en la taille de $AS(T) \Rightarrow O(n)$

- Algorithme séquentiel :
  - on rajoute dans l'arbre les suffixes les uns après les autres
  - l'ordre de rajout des suffixes dépend de l'algorithme considéré
  - gain de temps (par rapport à $O(n^2)$, cf. Algo Simple) dû à des sauts dans l'arbre en cours de construction.
    But : trouver rapidement l'endroit où construire le suffixe courant

L'Algorithme de McCreight (1976)

Algorithme qui insère les suffixes les uns après les autres, du plus long au plus court

Définitions :

- un préfixe commun à deux mots u et v = mot w qui est préfixe de u et de v
- le plus long préfixe commun à u et v = mot w préfixe commun de u et de v, et tel que $u[|w|+1] \neq v[|w|+1]$

  On notera

$$w = LCP(u, v)$$

L'Algorithme de McCreight

Propriété : Pour deux suffixes $s_1 = T[i..n]$ et $s_2 = T[j..n]$ d'un texte T, leur plus long préfixe commun correspond au mot codé par le chemin $r \rightarrow p$, où

- r est la racine de $AS(T)$
- p est le plus petit ancêtre commun dans $AS(T)$ entre les feuilles qui représentent s_1 et s_2 dans $AS(T)$

Définitions :

- Pour tout i, on appelle tete(i) le LCP de $T[i..n]$ et de $T[j..n]$, parmi tous les $j < i$
- Pour tout i, on appelle queue(i) le mot tel que $T[i..n] =$tete(i)$\cdot$queue(i)

L'Algorithme de McCreight

A chaque itération (= chaque insertion du suffixe $T[i..n]$ dans l'arbre) :

- on cherche tete(i) en suivant un chemin existant dans l'arbre
- on trouve le nœud qui correspond à tete(i) (ou on le crée s'il n'existe pas)
- on crée une fourche qui représente queue(i), et on numérote la feuille ainsi créée par i

Remarque : ce qui vient d'être expliqué n'est pas différent de ce qui est fait dans l'Algorithme Simple !

L'Algorithme de McCreight

Algorithme de McCreight :

- recherche rapide de tete(i) dans l'arbre en cours de construction
- par des sauts dans la lecture de l'arbre

Pour cela, deux concepts :

1. les liens suffixes
2. une propriété qui lie tete(i) à tete($i+1$)

Liens Suffixes

Construction, à chaque itération, de liens supplémentaires : les liens suffixes

Lien suffixe d'un nœud u vers un nœud u' ssi :

- u représente un mot $v = c \cdot v'$, $c \in \Sigma$ (càd, v commence par un caractère c , suivi d'un mot v') et
- u' représente le mot v'

On note

$$u' = s(u)$$

tete(i) et tete($i+1$)

Propriété : on fixe i, et on suppose que tete(i)=$T[i..i+k]$. Alors $T[i+1..i+k]$ est un préfixe de tete($i+1$).

Preuve :

- vrai si $k = 0$ (car $T[i+1..i+k] = \varepsilon$) $\Rightarrow$ supposons que $k > 0$
- on supposons que tete(i)=$c \cdot v$
- par définition de tete(i), il existe un $j < i$ tel que $LCP(i,j) = c \cdot v$
  Note : ici, $LCP(i,j)$ est un abus de notation pour dire $LCP(T[i..n], T[j..n])$

- $\Rightarrow$ les suffixes $T[i+1..n]$ et $T[j+1..n]$ partagent le même préfixe v
- par définition de tete($i+1$), v est un préfixe de tete($i+1$)

L'Algorithme de McCreight

- on sait donc que si tete(i)=$T[i..i+k]$, alors $T[i+1..i+k]$ est un préfixe de tete($i+1$)
- or $T[i+1..i+k] = s(T[i..i+k])$
- donc s(tete(i)) est un préfixe de tete($i+1$)
- vu dans l'arbre en cours de construction :

s(tete(i)) est un ancêtre de tete($i+1$)

Retour sur l'Algorithme de McCreight

Pour chaque itération i :

- trouver tete(i) (grâce à s(tete($i-1$)))
- créer la fourche
- insérer queue(i) et la feuille correspondante portant le numéro i
- construire le lien suffixe s(tete(i))

$\Rightarrow$ permet d'accélérer la recherche de l'endroit où se situe tete(i)

Analyse amortie de l'algorithme : construction en $O(n)$

En résumé sur l'Arbre des Suffixes

Construction Index	Recherche de Motifs	Espace Requis (Index)
$O(n)$	$O(m\sigma + occ)$	$O(n \log n)$

En résumé sur l'Arbre des Suffixes

- Complexité en temps : $O(n + m\sigma + occ)$
- Plus grossièrement, $occ \leq n - m + 1$ et $m \leq n \Rightarrow O(n)$
- Question : où est passé le $\log n$ dans la construction de $AS(T)$?

En résumé sur l'Arbre des Suffixes

Question : où est passé le $\log n$ dans la construction de $AS(T)$?

- ordinateur standard, entier stocké sur 32 bits (voire plus)
- $\rightarrow$ si $\log n \leq 32$ (càd $n \leq 4,3$ milliards environ), on considére $\log n$ comme une constante
- hypothèse appelée le "RAM model"
- $\Rightarrow$ dans le RAM model, la complexité de la construction de $AS(T)$ est en $O(n)$

Remarque 1 : on garde quand même le $\log n$ en espace, car permet de mieux comparer les choses
Ex : $AS(T)$ en $O(n \log n)$ alors que T est en $O(n \log \sigma)$

Remarque 2 : nous avons déjà utilisé cette hypothèse (sans le dire) pour complexité en espace de l'AFD reconnaissant $\Sigma^* P$

Sommaire

Idée générale

- stocker dans un tableau $TS[]$ (plutôt que dans un arbre) tous les suffixes possibles de T
- trier ces suffixes par ordre lexicographique
- faire de la recherche dichotomique de P dans T
- pas meilleur que la solution “arbre des suffixes”, mais nettement plus simple à implémenter !

Tableau des Suffixes

Intérêts principaux :

- [+] plus grande simplicité d'implémentation par rapport à l'arbre des suffixes
- [+] occupation mémoire moindre...
  - pas en terme de "grand O"
  - en moyenne un tableau des suffixes prend ∼4 à 5 fois moins de place qu'un arbre des suffixes

Complexité en espace

Tableau de taille n, contenant des entiers de 1 à $n \rightarrow O(n \log n)$

Ex. : Construction du tableau des suffixes $TS[]$ pour $T =$ `GCATCGCAGAGAGTATACAGTACG`

$TS[T] = [17, 22, 8, 10, 19, 12, 15, 3, 7, 18, 2, 23, 5, 24, 9, 11, 6, 1, 20, 13, 16, 21, 14, 4]$
car :

- $T[17..24] =$ `ACAGTACG`
- $T[22..24] =$ `ACG`
- $T[8..24] =$ `AGAGAGTATACAGTACG`
- $T[10..24] =$ `AGAGTATACAGTACG`
- $T[19..24] =$ `AGTACG`
- $T[12..24] =$ `AGTATACAGTACG`
- $T[15..24] =$ `ATACAGTACG`
- etc.

Recherche de motif – Méthode 1

- toutes les occurrences de P se trouvent dans un intervalle $I = TS[bg..bd]$ de $TS[]$
- But : trouver les bornes bg et bd de cet intervalle
- Trouver Borne gauche bg, puis Borne Droite bd $\rightarrow$ même méthode
- Recherche dichotomique pour trouver la Borne (Gauche ou Droite)
  - on continue même si on a trouvé P
  - on s'arrête quand l'intervalle de recherche est limité à 1 élément

Ex. : Recherche de $P = \texttt{GCAGAGAG}$ dans $T = \texttt{GCATCGCAGAGAGTATACAGTACG}$ en utilisant $TS[]$

Méthode 1 – Analyse

- $TS[]$ de taille $n \rightarrow \log n$ itérations
- chaque itération : jusqu'à m comparaisons de caractères
- $\rightarrow O(m \log n)$ pour trouver une borne
- $\rightarrow$ même complexité pour trouver les deux

$\Rightarrow O(m \log n)$

Recherche de motif – Méthode 2

- ajout d'une information pour accélérer les calculs
- information : LCP = Longest Common Prefix (cf. Arbres des Suffixes)
- tableau annexe stockant dans $LCP[i]$ la longueur du LCP entre les mots représentés par $TS[i-1]$ et $TS[i]$

Exemple : Tableau $LCP[]$ pour $T =$ `GCATCGCAGAGAGTATACAGTACG`

$TS[T] = [17, 22, 8, 10, 19, 12, 15, 3, 7, 18, 2, 23, 5, 24, 9, 11, 6, 1, 20, 13, 16, 21, 14, 4]$
et
$LCP[T] = [0, 2, 1, 4, 2, 4, ...]$
car :

- $T[17..24] =$ `ACAGTACG`
- $T[22..24] =$ `ACG`
- $T[8..24] =$ `AGAGAGTATACAGTACG`
- $T[10..24] =$ `AGAGTATACAGTACG`
- $T[19..24] =$ `AGTACG`
- $T[12..24] =$ `AGTATACAGTACG`
- $T[15..24] =$ `ATACAGTACG`
- etc.

Recherche de motif – Méthode 2

- [+] Permet d'accélérer la recherche de motif $\rightarrow$ passage de $O(m \log n)$ à $O(m + \log n)$
- [−] Construction du tableau LCP + algorithme de recherche difficiles à implémenter (si on veut garantir ces temps d'exécution)
- [−] Algorithme non décrit ici

Construction de l'index

Remarque 1 : partant d'un arbre des suffixes $AS(T)$, on peut construire $TS[]$ en faisant un parcours en profondeur de $AS(T)$

Exemple : avec $T = \texttt{CUICUI}$ ($TS[] = [4, 1, 6, 3, 5, 2]$)

Remarque 2 : parcours en profondeur d'un arbre est linéaire en sa taille

$\Rightarrow$ Un algorithme de construction possible :

1. construction de $AS(T)$
2. parcours en profondeur de $AS(T)$ pour produire $TS[]$

Analyse : 1. et 2. en $O(n)$ chacun $\Rightarrow$ construction du $TS[]$ en $O(n)$

Construction de l'index

- C'est de la triche !
- $\rightarrow$ $TS[]$ est utilisé pour éviter de construire l'arbre des suffixes
- d'autres algorithmes permettent de construire $TS[]$ en $O(n)$ sans passer par l'AS
- le tableau $LCP[]$ peut également être calculé en $O(n)$
- dépasse le cadre de ce cours faute de temps. Voir par exemple

  http ://www-igm.univ-mlv.fr/~mac/CHL/CHL-2011.pdf

  pour une description détaillée (et en français)
- vue inverse : partant du $TS[]$, on peut construire l'AS en temps linéaire (mais reste compliqué)

Complexité de la recherche de motif

	Construction Index	Recherche de Motifs	Espace Requis (Index)
Sans LCP	$O(n)$	$O(m \log n)$	$O(n \log n)$
Avec LCP	$O(n)$	$O(m + \log n)$	$O(n \log n)$

En résumé sur le tableau des suffixes

- Plus simple à implémenter que l'arbre des suffixes, du moins dans sa version sans LCP
- Espace équivalent à l'arbre des suffixes : $O(n \log n)$
- En pratique, $TS[]$ est plus compact que AS d'un facteur 4 ou 5 (capté dans le $O()$)

Sommaire

Conclusion

- présentation “express” de 3 structures à base d’index pour la recherche exacte de motifs
- Arbre des Suffixes et Tableau des Suffixes plus souvent utilisés que l’automate reconnaissant $\Sigma^* P$
- Très large littérature sur le sujet, très souvent technique. Voir par exemple :

  http ://www-igm.univ-mlv.fr/∼mac/CHL/CHL-2011.pdf

Conclusion

- de nombreux paramètres étudiés : taille de T et P, valeur de σ, espace mémoire théorique ($O(...)$) et pratique, tests sur des benchmarks, etc.
- applications biologiques
- variante naturelle : motifs avec erreurs, donc approchés
  - insertions
  - suppressions
  - *mismatches*