

Introduction à l'algorithmique du texte pour la bioinformatique

M1 Bioinformatique 2010-2011, 8h cours + 8h TD

Mathieu Raffinot

10 novembre 2010

Table des matières

1	Introduction	2
2	Recherche exacte	2
2.1	Cadre de la recherche d'un mot dans un texte	2
2.2	Recherche avant par l'automate Σ^*p	4
2.2.1	Algorithme de Morris-Pratt	5
2.2.2	Algorithme de Knuth-Morris-Pratt	9
2.2.3	Algorithme de Simon	10
2.2.4	Shift-And	12
2.3	Comment faire mieux ? La recherche arrière par facteurs	15
2.3.1	Approche BDM	16
2.3.2	BNDM	17
2.4	Recherche d'expression régulière	19
2.4.1	Parsing en arbre binaire	20
2.4.2	Stratégies de recherche exacte	21
2.4.3	Automate de Thompson	21
2.4.4	Recherche avec l'automate de Thompson	23
2.4.5	Sur le report des occurrences	23
3	Recherche approchée	24
3.1	Distance de Levenshtein	24
3.2	Recherche avec la distance de Levenshtein	27
3.3	Needleman-Wunsch, alignement global	27
3.4	Smith-Waterman, alignement local	29
3.5	BLAST et FASTA	30
4	Indexation de texte	31
4.1	Table des suffixes	31
4.1.1	Comment faire une recherche d'un motif sur cette permutation ?	32
4.1.2	Comment construire cette table ?	32
4.2	Arbre des suffixes	33
4.2.1	Comment construire l'arbre des suffixes ?	34

1 Introduction

Les techniques d'algorithmique du texte sont très utilisées en bioinformatique pour la manipulation "virtuelle" (par ordinateur) de séquences de nucléotides ou d'acides aminées. Nous en présentons quelques unes en essayant de transmettre les intuitions nécessaires à l'utilisation et à la création d'algorithmes dans ce domaine. Nous commençons par des algorithmes de recherche d'un mot dans un texte. C'est un problème qui a l'air simple mais qui en réalité est beaucoup plus complexe qu'il n'y paraît si nous voulons le résoudre de la manière la plus efficace possible. Ensuite, nous verrons comment rechercher une expression régulière. Toutes ces recherches sont exactes. Cependant en bioinformatique, de nombreuses recherches sont approchées, mais les algorithmes les plus efficaces sont souvent basés sur des recherches exactes. Nous verrons les algorithmes de base de recherches approchées. Enfin, nous aborderons sans l'approfondir la question de l'indexation d'un texte pour y faire un grand nombre de recherches plus efficacement.

2 Recherche exacte

Nous recherchons un mot $p = p_1 \dots p_m$ dans un texte (ou séquence) $t = t_1 \dots t_n$ qui sont tous les deux des suites de caractères appartenant à un alphabet Σ fixé. Nous supposons toujours par la suite $n \geq m$, sinon le problème a peu de sens ! Nous notons Σ^* l'ensemble des mots sur l'alphabet Σ .

2.1 Cadre de la recherche d'un mot dans un texte

L'algorithme naïf consiste simplement à rechercher le mot à chaque position possible dans le texte. Pour chaque position du texte, nous comparons caractère contre caractère de gauche à droite le texte et le motif et nous continuons ainsi tant que les lettres sont les mêmes. Si nous arrivons à la fin du motif, nous marquons une occurrence. Le pseudo-code de cette algorithme naïf est donné figure 1.

```
Algorithme_naïf( $P = p_1 p_2 \dots p_m, T = t_1 t_2 \dots t_n$ )
1.  Pour  $i$  allant de 1 à  $\dots n - m + 1$  Faire
2.     $j \leftarrow 1$ 
3.    Tant que ( $j \leq m$  ET  $t_{i+j} = p_j$ ) Faire
4.       $j \leftarrow j + 1$ 
5.    Fin du tant que
6.    Si  $j = m + 1$  Faire
7.      report d'une occurrence en  $i$ 
8.    Fin du si
9.  Fin du pour
```

FIG. 1 – Algorithme naïf. A la ligne 3 le test $t_i \neq p_j$ est supposé n'être fait que si le $j \leq m$ est VRAI.

L'étude de l'algorithme naïf va nous donner un cadre sur le problème de la recherche d'un mot dans un texte et nous permettre de spécifier les bornes de complexité à améliorer.

La première question qui se pose est : quelle est sa complexité dans le pire des cas, en nombre de comparaisons ? Quel que soit le texte, nous le parcourons (quasiment) entièrement par la boucle ligne 1. Pour chaque position, dans le pire des cas, nous parcourons tout le motif et effectuons m comparaisons. Au total, nous pouvons donc effectuer nm comparaisons et la complexité dans le pire des cas est donc de $O(nm)$.

Estimons maintenant sa complexité moyenne, dans le modèle de probabilité le plus simple possible. Nous considérons que la probabilité d'apparition d'un caractère à une position donnée du texte ou du motif est indépendante des autres caractères du mot (modèle dit de Bernoulli), et que de plus la probabilité d'apparition d'un caractère à une position donnée est la même pour tous les caractères de l'alphabet (Bernoulli équiprobable), et donc de $1/|\Sigma|$.

Pour chaque position i du texte, la probabilité que le caractère du texte t_i soit le même que le premier caractère p_1 du motif est de $1/|\Sigma|$. Dans ce cas, nous continuons la comparaison avec le deuxième caractère. La probabilité qu'ils soit égaux est la même avec une probabilité de $1/|\Sigma|$. Et ainsi de suite jusqu'au $m^{\text{ème}}$ caractère. La probabilité de lire k caractères est donc la probabilité d'en lire un, puis de lire le deuxième, puis de lire le suivant et ainsi de suite jusqu'au $k^{\text{ème}}$. Nous devons donc multiplier les probabilités et nous lisons k caractères avec une probabilité de $\frac{1}{|\Sigma|^k}$. La moyenne (ou l'espérance) des caractères lus pour une position donnée est de $\sum_{i=1}^k \frac{k}{|\Sigma|^k}$ et donc la complexité moyenne est bornée par $O(n \sum_{i=1}^k \frac{k}{|\Sigma|^k})$. Ce n'est pas très parlant !

Intéressons nous à la série $\sum_{i=1}^m \frac{k}{|\Sigma|^k}$. Comment évolue t'elle lorsque la taille du motif augmente ? Prenons par exemple $|\Sigma| = 4$. Nous obtenons pour un motif de taille 4 la somme $1/4 + 2/16 + 3/64 + 4/256 = 0,4375$, pour un motif de taille 5 nous avons $0,4424$, etc. Pour y voir plus clair traçons la courbe de la figure 2.

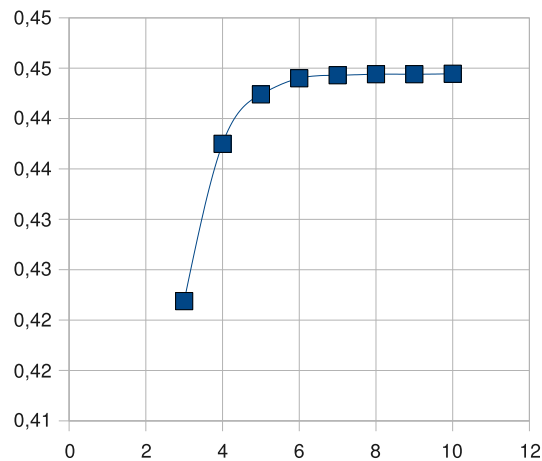


FIG. 2 – Valeurs de la série $\sum_{i=1}^k \frac{k}{|\Sigma|^k}$ pour $|\Sigma| = 4$ et k allant de 1 à 10.

La série semble converger, c'est-à-dire qu'elle semble bornée par une constante (dépendante de $|\Sigma|$). Confirmons le par le test de Dirichlet : soit une série $\sum_{k=1}^{\infty} U_k$; si $U_{k+1}/U_k < 1$, alors la série converge. Dans notre cas,

$$\frac{U_{k+1}}{U_k} = \frac{(k+1)/|\Sigma|^{k+1}}{k/|\Sigma|^k} = \frac{k+1}{k|\Sigma|}.$$

Dès que $|\Sigma| > 1$, $(k+1)/(k|\Sigma|) < 1$ pour tout $k \geq 2$ et la série converge ! Il existe une

constante K (dont la valeur peut être rendue indépendante de $|\Sigma|$) telle que $\sum_{i=1}^m \frac{k}{|\Sigma|^k} \leq K$ quel que soit la taille m du motif! Et la complexité moyenne de tout l'algorithme est donc bornée par $O(Kn)$, donc $O(n)$. L'algorithme naïf est linéaire en moyenne!

Nous nous intéressons donc à un problème de recherche d'un mot dans un texte que nous savons déjà résoudre facilement en $O(nm)$ dans le pire des cas et $O(n)$ en moyenne. Commençons par améliorer notre approche pour avoir des algorithmes linéaires dans le pire des cas. L'approche naturelle est d'essayer de lire les caractères du texte les uns après les autres en essayant de garder suffisamment d'information d'une position sur l'autre pour ne pas tout recalculer à zéro à chaque nouvelle position.

2.2 Recherche avant par l'automate Σ^*p

Il existe en fait plusieurs manières de rechercher un mot p dans un texte en lisant les caractères du texte les uns après les autres, les deux principales étant le hachage (et principalement l'algorithme Karp-Rabin, que nous ne verrons pas) et la lecture du texte au travers d'un automate reconnaissant le langage Σ^*p . C'est cette dernière que nous allons approfondir ici.

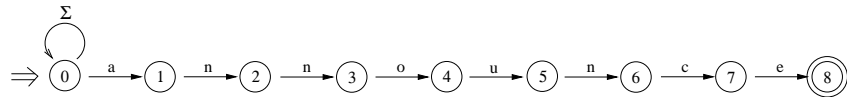


FIG. 3 – Automate non déterministe qui reconnaît tous les préfixes du mot “anas”.

Prenons comme exemple le mot *anas* que nous rechercherons dans un texte. Pour construire un automate non déterministe qui recherche ce mot dans un texte, il suffit de construire l'automate de la figure 3. La boucle sur l'état initial 0 étiquetée par Σ permet de toujours recommencer la recherche à zéro lorsque nous lisons un caractère du texte. Ensuite, le mot est trouvé si nous atteignons l'état final, c'est-à-dire si nous avons suivi le chemin de l'état 0 à l'état 6. Notons au passage qu'après chaque lecture d'un caractère du texte plusieurs états peuvent être *actifs*, c'est-à-dire que nous y sommes arrivés et que l'on peut potentiellement en partir à la lecture du caractère suivant. Rechercher un motif représenté directement avec son automate non déterministe demande un temps de $O(nm)$ car il faut gérer une liste de tous les états actifs.

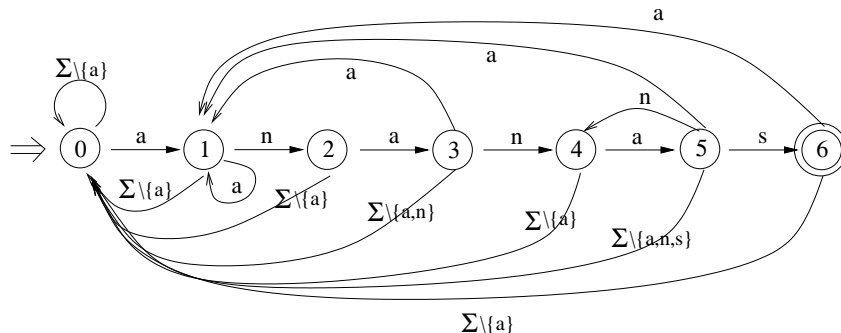


FIG. 4 – Automate déterministe qui reconnaît tous les préfixes du mot “anas”.

Cet automate non déterministe est bien sûr déterminisable. Sur notre exemple, nous ob-

tenons l'automate de la figure 4. La recherche avec cet automate serait de l'ordre de $O(n)$ en utilisant une mémoire de taille $m|\Sigma|$. Le problème vient de cette taille et de son temps de construction, qui sont bien sûr au minimum de l'ordre de $m|\Sigma|$. Nous allons étudier de suite plusieurs techniques pour garder une recherche linéaire dans le pire des cas tout en gardant un précalcul sur le motif linéaire en sa taille en temps et en espace.

2.2.1 Algorithme de Morris-Pratt

Le premier algorithme de recherche d'un mot linéaire (en $O(m+n)$) dans le pire des cas est apparu en 1970 [13]. C'est l'algorithme de Morris-Pratt (MP) qui simule le fonctionnement de l'automate $\mathcal{A}(p)$ au moyen d'une fonction de suppléance qui utilise la notion qui suit.

Définition 2.1 Soit $v \in \Sigma^* \setminus \{\epsilon\}$, $b \in \Sigma^*$ est un bord de v si b est à la fois préfixe et suffixe de v .

L'algorithme MP fait un usage intensif de la propriété des bords suivante.

Proposition 2.1 Soit $v \in \Sigma^* \setminus \{\epsilon\}$, $b_1, b_2 \in \Sigma^*$ deux bords de v tel que $|b_1| < |b_2|$. Alors b_1 est un bord de b_2 .

Preuve b_1 et b_2 sont des préfixes de v . Comme $|b_1| < |b_2|$, b_1 est un préfixe de b_2 . De même, b_1 et b_2 sont des suffixes de v , et comme $|b_1| < |b_2|$, b_1 est aussi un suffixe de b_2 . ■

Supposons maintenant que pour chaque préfixe du mot p nous ayons calculé la taille de son plus long bord dans une table **Bord**, en utilisant la valeur arbitraire -1 pour le préfixe vide ϵ . Par exemple, pour le mot **ananas**, la table des bords est :

Long. Préf.	0	1	2	3	4	5	6
Taille du plus long bord	-1	0	0	1	2	3	0

L'algorithme MP maintient pour chaque position i du texte le plus long suffixe de $t_1 \dots t_i$ qui est aussi un préfixe du mot p recherché.

Supposons qu'à la position i nous connaissions la taille $j(i)$ de ce plus long suffixe. Comment la calculer pour la position $i+1$?

Si $t_{i+1} = p_{j(i)+1}$, c'est-à-dire si le caractère du texte lu t_{i+1} est égal au caractère $p_{j(i)+1}$ qui suit dans p ce plus long suffixe qui est aussi un préfixe de p , alors $p_1 \dots p_{j(i)+1}$ est un préfixe de p qui est suffixe de $t_1 \dots t_i t_{i+1}$. C'est en fait le plus long qui vérifie cette propriété (car sinon il y en aurait eu un plus long que $j(i)$ à la position i) et donc dans ce cas $j(i+1) \leftarrow j(i) + 1$.

Dans le cas contraire, si $t_{i+1} \neq p_{j(i)+1}$, l'algorithme MP décale le motif pour aligner le suffixe du texte $t_1 \dots t_i$ avec le plus long bord de $p_1 \dots p_{j(i)}$. Notons b la taille de ce bord. Le caractère t_{i+1} est alors comparé avec p_{b+1} . S'il est égal, $j(i+1) \leftarrow b+1$. Dans le cas contraire, nous considérons le plus long bord suivant de $j(i)$, qui est en fait, par la propriété 2.1, le plus long bord du préfixe $p_1 \dots p_b$. Nous recommençons ainsi jusqu'à ce que (a) soit on ait trouvé un bord qui est suivi par le bon caractère, (b) soit ce bord n'existe pas. Dans le cas (a) la valeur de $j(i+1)$ est connue. Dans le deuxième cas (b) $j(i+1) \leftarrow 0$. La figure 5 illustre ce décalage.

D'un point de vue plus technique, le bord ϵ est un bord pour tous les préfixes, il sera testé comme taille de bord dans tous les cas si un plus grand bord b qui satisfait $t_{i+1} = p_{b+1}$ n'est

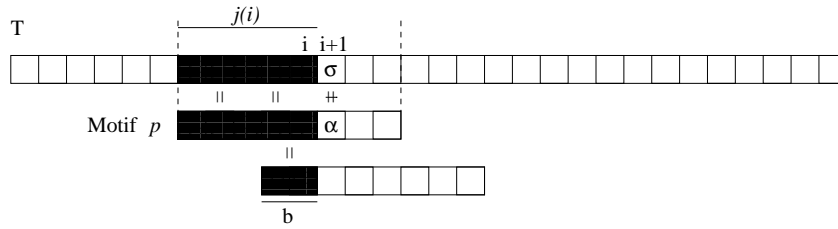


FIG. 5 – Le déplacement de l’algorithme **Morris-Pratt**. Le préfixe de p de taille b est un suffixe du préfixe $p_1 \dots p_{j(i)}$ mais en est aussi un préfixe. Le caractère $\alpha = p_{j(i)+1}$ est différent de $\sigma = t_{i+1}$.

pas trouvé avant. Dans ce cas, si $p_1 = t_{i+1}$, on est dans le cas (a), sinon $b \leftarrow \text{bord}(\varepsilon) = -1$. Notre test d’arrêt pour le cas (b) doit donc tester simplement si $b \neq -1$.

Ecrivons le pseudo-code du MP en supposant que la table $\text{Bord}[0..m]$ est déjà calculée. La figure 6 en présente une version.

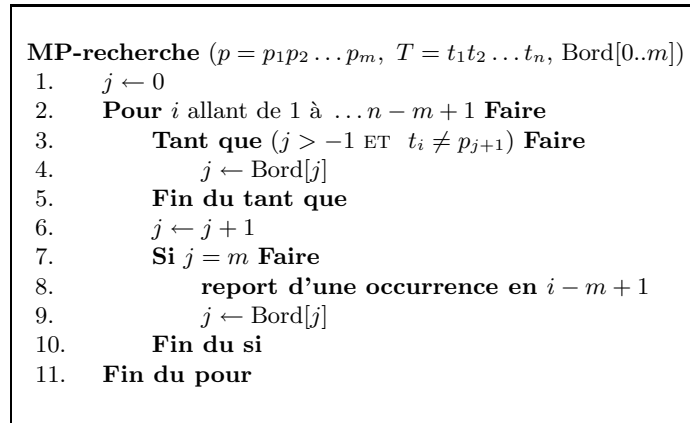


FIG. 6 – Algorithme **MP**, phase de recherche. A la ligne 3 le test $t_i \neq p_{j+1}$ est supposé n’être fait que si le $j > -1$ est VRAI.

Il nous faut maintenant prouver la validité de l’algorithme MP-recherche, c’est-à-dire qu’il trouve bien toutes les occurrences de p dans t et elles seules. Nous supposons que la table Bord est donnée et que ses valeurs sont justes.

Théorème 2.2 *L’algorithme MP-recherche est valide*

Preuve Nous supposons que la table des bords est valide. Il n’est pas dur de vérifier que l’algorithme fonctionne pour $i = 1$. Supposons maintenant qu’il existe au moins une position i du texte telle que $j(i)$ ne soit pas égal à la taille $j'(i)$ du plus long suffixe de $t_1 \dots t_i$ qui soit aussi un préfixe de p . De toutes ces positions, prenons la plus petite. Le calcul de $j(i - 1)$ à la position précédente est correct. Intéressons nous au passage de $j(i - 1)$ à $j(i)$.

Si $t_i = p_{|j(i-1)|+1}$, l’algorithme fixe $j(i) \leftarrow j(i - 1) + 1$. S’il existait un plus long suffixe du texte préfixe de p , ce serait vrai pour la position précédente $i - 1$ et $j(i - 1)$ ne serait pas correct, ce qui contredit l’hypothèse.

La différence entre $j(i)$ et $j'(i)$ ne peut donc provenir que du deuxième cas lorsque nous parcourons la table des bords. Supposons que $|j'(i)| > |j(i)|$. Nous avons $|j'(i) - 1| < |j(i - 1)|$ et comme $p_1 \dots p_{j'(i)}$ est un suffixe de $t_1 \dots t_i$, $p_1 \dots p_{j'(i)-1}$ est un suffixe de $t_1 \dots t_{i-1}$ préfixe de p de taille plus petite que $j(i - 1)$. Donc $p_1 \dots p_{j'(i)-1}$ est un bord de $p_1 \dots p_{j(i)}$. En outre, $t_i = p_{j'(i)}$. En parcourant la liste des bords par taille du plus grand bord de $j(i - 1)$ pour identifier le plus grand qui soit suivi de t_i , nous devons obligatoirement tester $p_1 \dots p_{j'(i)-1}$ et fixer $j(i)$ à $j'(i) - 1 + 1$. D'où la contradiction.

Supposons maintenant que $|j'(i)| < |j(i)|$. Les opérations que nous faisons font que la valeur calculée en $j(i)$ correspond bien à la taille d'un suffixe de $t_1 \dots t_i$ qui est aussi préfixe de p . Nous obtenons alors directement une contradiction sur le fait que $j'(i)$ soit la taille du plus grand! ■

Avant de nous intéresser à la construction de la table des bords, essayons d'identifier les liens entre le MP et l'automate $\mathcal{A}(p)$. Comme un bord est un préfixe associé à un autre préfixe et que chaque préfixe correspond à un état dans $\mathcal{A}(p)$, nous pouvons dessiner la table Bord sous forme de flèches arrières sur l'automate. Sur notre exemple courant nous obtenons la figure 7.

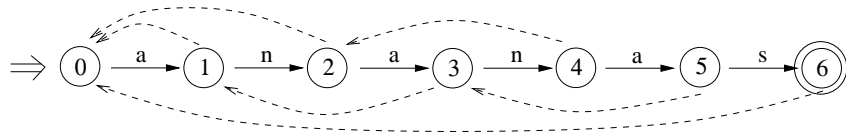


FIG. 7 – Codage de Morris-Pratt avec une fonction de suppléance représentée sous forme de flèches arrières pointillées.

Intéressons nous à la transition de l'état 5 à 1 par a dans $\mathcal{A}(p)$. Pour la calculer à partir des bords, sur la figure 7, il faut passer de 5 à l'état 3 par la transition arrière de 5 à 3 qui correspond au plus long bord de $anana$, *i.e.* ana , puis, comme l'état 3 n'a pas de transition sortante par a , nous utilisons de nouveau un lien arrière de 3 à 0 (correspondant au bord a de ana). Comme l'état 0 a une transition sortante par a , nous devons aller par a de l'état 5 à 1 dans $\mathcal{A}(p)$. Il est évidemment possible de généraliser cette technique et ainsi de calculer l'automate $\mathcal{A}(p)$ à partir de la table des bords, qui est donc en fait une représentation compacte de l'automate $\mathcal{A}(p)$.

Parcourons maintenant notre texte T au moyen de l'algorithme MP-recherche et regardons le évoluer sur l'automate 7, c'est-à-dire la table des bords représentée sous forme d'automate. Prenons un exemple, essayons de lire *yvananas*. Plaçons nous sur l'état 0. La lecture de y ne change rien, dans le pseudo-code (figure 6) le test $y = t_1 \neq p_1 = a$ passe et j reçoit $Bord[0] = -1$, ce qui bloque la boucle au test suivant. Et ensuite ligne 6 $j \leftarrow j + 1 = 0$ et nous recommençons ligne 2 avec $j(2)$ qui vaut 0. La lecture de v provoque la même chose et nous recommençons ligne 2 avec $j(3) = 0$. La lecture de a fait échouer le test ligne 2 car $a = t_3 = p_1 = a$ et $j \leftarrow 1$ ligne 6. Sur l'automate nous sommes passé de l'état 0 à l'état 1 par la transition étiquetée par a . Nous lisons n et nous passons de l'état 1 à l'état 2 par la transition avant étiquetée par n et $j(5) \leftarrow 2$ ligne 6. Nous continuons ainsi en lisant successivement chaque lettre de ana et nous arrivons à l'état 5 avec $j(8) \leftarrow 5$.

Nous lisons maintenant n . Comme $a = t_9 \neq p_6 = s$, j reçoit ligne 4 $Bord[5]$ égal à 3 et nous revenons au test de la boucle **Tant que** ligne 3. Comme $a = t_9 = p_4 = a$, la boucle s'arrête et nous passons ligne 6 à $j(9) \leftarrow 4$. Qu'avons nous fait sur l'automate? Nous avons été bloqué à l'état 5 car nous n'avons pas trouvé de transition sortante étiquetée par $t_9 = a$.

Nous sommes donc passé au bord de 5, c'est-à-dire à l'état 3, et nous avons recherché de nouveau à partir de cet état une transition par a . Comme l'état 3 en a une qui mène à 4, nous sommes arrivés dans l'état 4.

Continuons. Nous lisons $t_{10} = a$. Comme le $t_{10} = p_5 = a$, la boucle **Tant que** ligne 3 s'arrête tout de suite et ligne 6 $j(10) \leftarrow j(9) + 1 = 5$. Sur l'automate, comme l'état 4 a une transition sortante étiquetée par a , nous la suivons et arrivons dans l'état 5. Nous lisons maintenant $t_{11} = s = p_6$. Comme pour la lettre précédente, la boucle **Tant que** s'arrête de suite et ligne 6 $j(11) \leftarrow j(10) + 1 = 6$. Comme ligne 7 $j(11) = 6 = m$, nous entrons dans le corps du *Si* ligne 8. Nous reportons donc une occurrence du motif en position 5 et ligne 10 $j(11) \leftarrow \text{Bord}[6] = 0$. Dans l'automate, nous passons directement de l'état 5 à l'état 6 par la transition étiquetée par s . Comme l'état 6 est final, nous reportons une occurrence. Ensuite, nous revenons au bord du motif lui-même par la transition arrière partant de 6 qui mène à l'état 0.

Que constatons nous à partir de cet exemple ? La table des bords permet en fait de simuler le fonctionnement de l'automate de recherche $\mathcal{A}(p)$ en occupant un espace linéaire en mémoire. C'est donc une représentation compressée de l'automate $\mathcal{A}(p)$!

Cette observation va nous permettre d'établir intuitivement le résultat principal de l'algorithme MP-recherche, c'est-à-dire sa linéarité en la taille du texte. Lorsque nous lisons un caractère du texte dans l'automate représentant la table des bords (figure 7), nous pouvons soit (a) rester sur place à l'état 0, (b) avancer au moyen des transitions avant si le caractère lu correspond à celui du motif, et (c) reculer de transition arrière en transition arrière (donc de bord en bord) jusqu'à l'état 0. Et ainsi de suite, pour chaque caractère du texte. Evaluons maintenant le nombre de transitions avant et arrière que nous pouvons effectuer en lisant le texte. Pour reculer d'une transition arrière, il faut au préalable avoir avancé dans l'automate d'au moins ... une transition avant ! Donc globalement sur le texte, nous ne pouvons pas reculer plus que nous avons parcouru de transition avant. Comme pour parcourir une transition avant il faut lire un caractère du texte, nous ne pouvons pas reculer au total sur tout le texte de taille n plus de n fois ! Le nombre de comparaisons du MP-recherche est donc borné au total par le nombre n de caractères lus plus le nombre de retours arrière borné par n , et donc par $2n$. L'algorithme MP-recherche est bien linéaire en la taille du texte ! Nous venons de prouver le théorème suivant.

Théorème 2.3 *L'algorithme MP-recherche fonctionne en temps $O(n)$.*

Il ne nous reste plus qu'à calculer la table des bords ! Pour bien voir ce qui se passe, essayons de construire l'automate de la figure 7 de gauche à droite, caractère après caractère, comme sur la figure 8.

Partons d'un état initial 0 et commençons par ajouter a . Nous créons l'état 1 et une transition de 0 vers 1 étiquetée a . Le bord de l'état 1 correspond au bord du préfixe a , et est donc de longueur nulle. Créons une flèche arrière pointillée de 1 vers 0. Nous obtenons l'automate de la figure 8-(a). Ajoutons n . Nous créons l'état 2 et une transition de l'état 1 vers l'état 2 par n . Le bord de l'état 2 correspond au bord du préfixe an , qui est en fait le plus long bord du préfixe précédent, de a , qui soit suivi de n s'il existe. Comme il n'existe pas, nous créons une flèche arrière pointillée de 2 vers 0. Ajoutons maintenant a . Nous créons un état 3 et une transition de 2 à 3 par a . Maintenant, pour calculer le plus long bord de ana , nous testons si le plus long bord de an , c'est-à-dire ϵ est suivi de a . Pour le faire, il suffit de suivre la transition arrière de l'état correspondant à an , donc l'état 2, de suivre son lien

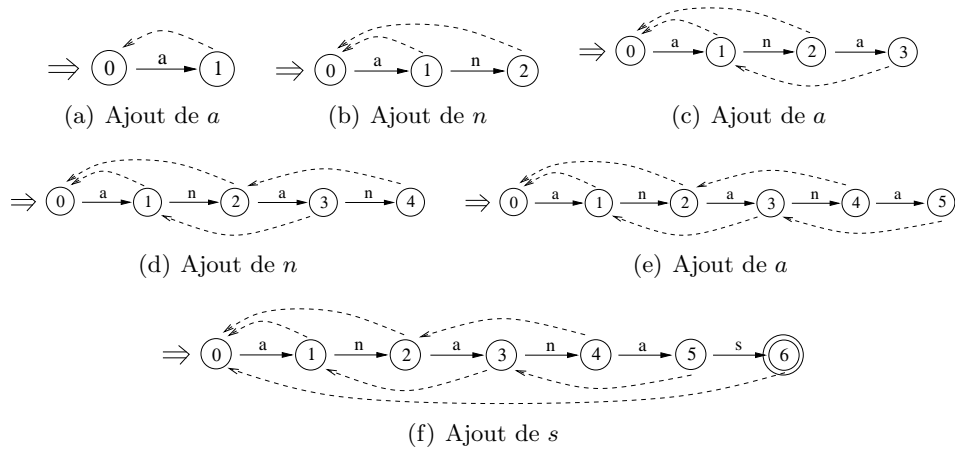


FIG. 8 – Construction de l’automate des bords de **ananas** caractère après caractère.

arrière qui arrive en l’état 0, et de regarder s’il existe une transition sortante de 0 par a . C’est le cas, la transition mène à l’état 1, et donc nous créons une flèche arrière pointillée de l’état 3 vers 1. Ajoutons n , nous créons un état 4 et une transition de 3 à 4 par n . Pour calculer son plus long bord, nous suivons l’arête arrière qui part de 3 vers 1. comme l’état 1 est suivi de n , nous créons une flèche arrière de 4 vers 2. De la même manière, pour le a suivant nous créons un état 5 et une transition de 4 vers 5 par a . Suivons le lien arrière partant de 4. Il arrive en 2, qui possède une transition sortante vers 3 étiquetée par a . Nous créons donc un lien arrière de 5 vers 3. Ajoutons la dernière lettre s . Nous créons un état 6 final et une transition de 5 vers 6 étiquetée par s . Calculons son plus grand bord. Le plus grand bord de 5 est 3. Mais 3 n’est pas suivi d’une transition par s . Nous sautons donc au plus grand bord suivant, en suivant le lien arrière qui part de 3. Il mène à 1, mais 1 n’est pas non plus suivi d’une transition par s . Nous ”sautons” au plus grand bord suivant, en suivant la transition arrière partant de 1. Elle arrive en 0. Comme 0 ne possède pas de transition sortante par s , nous devons considérer le plus grand bord de 0. Comme il a été fixé à une valeur ”sentinelle” -1 , nous arrêtons le calcul et le plus grand bord de l’état 6 est fixé à 0.

Ouf, c’est fastidieux mais nécessaire pour bien comprendre l’algorithme. Que remarquons nous? L’algorithme de construction de la table des bords ressemble beaucoup à l’algorithme de recherche lui-même! C’est en fait la même idée, mais appliquée à la recherche du motif dans le motif lui-même. Ecrivons le pseudo-code qui correspond, donné figure 9.

Comme l’algorithme de calcul de la table des bords est très proche de l’algorithme de recherche appliqué sur le motif lui-même, la même astuce d’analyse permet de borner sa complexité dans le pire des cas en nombre de comparaisons, et nous obtenons directement le théorème suivant.

Théorème 2.4 *L’algorithme MP-précalcul fonctionne en temps $O(m)$.*

2.2.2 Algorithme de Knuth-Morris-Pratt

L’algorithme de Knuth-Morris-Pratt, publié en 1977 [9], est une version améliorée de l’algorithme MP en ajoutant une astuce qui se situe seulement au moment du calcul de la table des bords. La phase de recherche est la même.

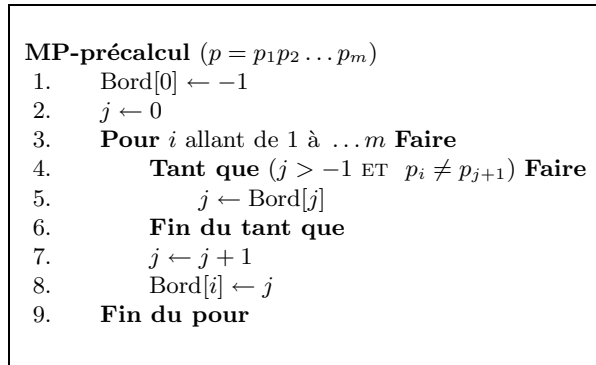


FIG. 9 – Algorithme **MP**, phase de précalcul. A la ligne le test $p_i \neq p_{j+1}$ est supposé n’être fait que si le $j > -1$ est VRAI.

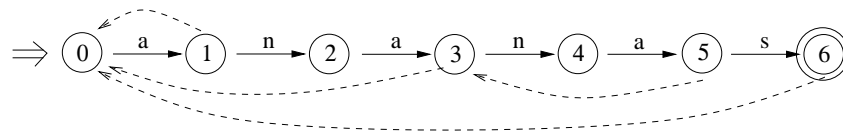


FIG. 10 – Codage de Knuth-Morris-Pratt avec une fonction de suppléance représentée sous forme de flèches arrières pointillées.

Supposons que dans la ligne 3 de l’algorithme **MP-recherche** nous échouions parce que $t_i \neq p_{j+1}$. Nous considérons maintenant le bord de taille b du préfixe de taille j de p . Si $p_{b+1} = p_{j+1}$, il est clair que la nouvelle comparaison de t_i contre p_{b+1} va échouer! Nous pouvons donc l’économiser en précalculant des bords qui sont suivis de caractères différents (sauf pour le motif lui-même). A partir de l’automate de la figure 7 qui représente les bords calculés par **MP**, nous obtenons les nouveaux bords représentés sur la figure 10. Nous nommons ces bords les *kmp-bords*.

La modification du pseudo_code de **MP-précalcul** pour obtenir la nouvelle table des *kmp-bords* est relativement simple. Le nouveau pseudo_code est donné figure 11. Nous remontons de bord en *kmp-bord* pour calculer le plus long *kmp-bord*. Si c’est du motif lui-même dont nous calculons le *kmp-bord* (test $i = m$ ligne 8), alors le plus long bord est valide, c’est bien le *kmp-bord* recherché. Sinon nous testons le caractère qui suit ce plus long bord et le comparons avec le caractère qui suit le préfixe dont nous sommes en train de calculer le bord. S’il sont différents (test ligne 8) nous avons trouvé le plus long *kmp-bord* (ligne 9). S’il sont égaux, il suffit de prendre le plus long *kmp-bord* de ce bord (ligne 11).

2.2.3 Algorithme de Simon

L’algorithme de Simon [5] est basé sur une observation fine sur l’automate de recherche $\mathcal{A}(p)$ qui possède en fait une propriété très forte : le nombre de transitions arrières qui arrivent sur un état autre que l’état initial est borné par $m = |p|$! Pour obtenir un algorithme efficace de recherche avant qui occupe un espace de mémoire en $O(m)$, il suffit donc en fait de coder cet automate sans les transitions arrières qui mènent sur l’état 0. L’automate que nous devons coder pour rechercher le motif *ananas* est donné figure 12.

D’où vient cette propriété? Dans ce paragraphe, une transition arrière signifie une transition

```

KMP-précalcul ( $p = p_1 p_2 \dots p_m$ )
1.   $\text{KMPBord}[0] \leftarrow -1$ 
2.   $j \leftarrow 0$ 
3.  Pour  $i$  allant de 1 à  $\dots m$  Faire
4.      Tant que ( $j > -1$  ET  $p_i \neq p_{j+1}$ ) Faire
5.           $j \leftarrow \text{KMPBord}[j]$ 
6.      Fin du tant que
7.       $j \leftarrow j + 1$ 
8.      Si ( $i = m$  OU  $p_{i+1} \neq p_{j+1}$ ) Faire
9.           $\text{KMPBord}[i] \leftarrow j$ 
10.     Sinon
11.          $\text{KMPBord}[i] \leftarrow \text{KMPBord}[j]$ 
12.     Fin du si-sinon
13. Fin du pour

```

FIG. 11 – Algorithme **KMP**, phase de précalcul. A la ligne le test $p_i \neq p_{j+1}$ est supposé n’être fait que si le $j > -1$ est VRAI.

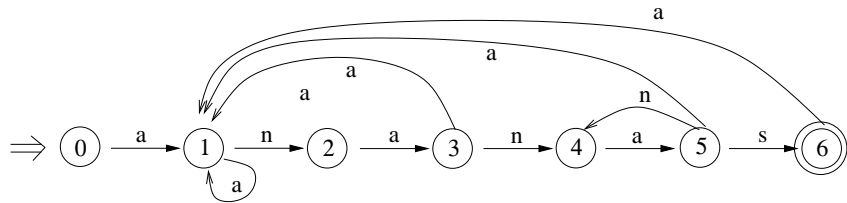


FIG. 12 – Codage de Simon.

arrière qui n’arrive pas dans l’état initial 0. Regardons la figure 13. Nous avons une transition arrière étiquetée par α partant d’un préfixe que nous notons v . Le mot u est un bord v et v est suivi par $\beta \neq \alpha$ dans p . Notons $dist$ la distance de la fin de v à la fin du préfixe de p de taille u . Il ne peut n’y avoir que m valeurs distinctes de $dist$. Supposons maintenant qu’il existe une deuxième transition arrière distincte menant à la même valeur de $dist$. Supposons dans un premier temps que cette transition parte de plus loin, comme sur la figure 13. A ce moment là, comme $u\alpha$ est un préfixe du motif p et que la deuxième transition arrière est aussi construite à partir d’un bord, une deuxième occurrence de $u\alpha$ devrait se trouver exactement à la même place que le $u\beta$ qui a mené à la première transition. Ce qui n’est pas possible ! Si maintenant nous supposons que cette deuxième transition parte de moins loin que la première, nous obtenons une impossibilité symétrique avec le bord du préfixe qui est à l’origine de cette transition. Dans les deux cas nous obtenons une contradiction, et par conséquent il ne peut exister qu’une seule transition arrière pour un $dist$ fixé, et donc au plus m transitions arrières !

Pour implémenter cet automate, il semble qu’il suffise simplement de coder ses transitions avant et les transitions arrières qui n’arrivent pas sur l’état initial. Mais il faut cependant faire attention. Si pour chaque état nous codons ses transitions dans n’importe quel ordre, nous obtenons une recherche en temps $O(n|\Sigma|)$ dans le pire des cas ! Pour rendre cette recherche plus efficace, il faut s’inspirer de l’algorithme MP et KMP et trier pour chaque état ses transitions par distance croissante à cet état, ce qui est l’analogie des plus long bords croissants. Dans ce cas, nous pouvons réutiliser l’astuce d’amortissement de la complexité de l’algorithme MP

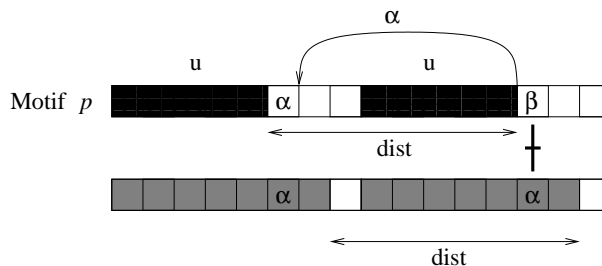


FIG. 13 – La propriété de $\mathcal{A}(p)$ utilisée par l’algorithme de Simon. Pour une même valeur de $dist$ il ne peut y avoir qu’une seule transition arrière qui n’arrive pas dans l’état initial.

et la recherche devient linéaire en $O(n)$.

2.2.4 Shift-And

Le processeur possède un certain nombre d’emplacements mémoire interne appelés registres. Ils ont une taille notée w qui maintenant varie entre 16 et 64 bits et qui vaut le plus souvent 32 sur les processeurs courants. Le processeur permet un certain nombre d’opérations sur ces registres, dont les plus courantes sont le $+$, $-$, $\%$, $*$. Cependant il en existe d’autres un peu moins usités comme ceux de la table qui suit (les exemples sont faits sur des registres virtuels de taille 4).

Opérateur	Description	Exemple
$\&$	ET binaire bit à bit	$(1100)\&(1010) = (1000)$
$ $	OU binaire bit à bit	$(1100)\ (1010) = (1110)$
\sim	NOT binaire bit à bit	$\sim(1100) = (0011)$
$\ll x$	décalage à gauche de x bits	$(1100)\ll 1 = (1000)$

Le shift-and [3, 16] utilise ces opérateurs pour simuler l’automate de recherche $\mathcal{A}(p)$. Il est particulièrement efficace sur des recherches de mot de petite taille (inférieure à la taille du registre) mais continue d’être efficace même sur des tailles plus grandes en simulant un registre de grande taille au moyen de plusieurs registres de taille inférieure.

Précalcul. A chaque caractère $\alpha \in \Sigma$ de l’alphabet nous associons un mot machine $B[\alpha] = b_{m-1} \dots b_0$ tel que $b_i = 1$ si et seulement si $p_{i+1} = \alpha$. Par exemple, pour le mot *ananas*, nous obtenons $B[a] = 010101$.

Recherche. Nous stockons dans un registre D l’ensemble des valeurs courantes (actif = 1, passif = 0) des états de l’automate de recherche non déterministe lus à l’envers (la raison de ce renversement vient du shift “ \ll ” sur la gauche). Par exemple, après la lecture de *yana* dans l’automate non déterministe de recherche avant du mot *ananas*, $D = 000101$. Lors de la lecture d’un caractère α du texte, D est mis à jour de la manière suivante :

$$D \leftarrow ((D \ll 1) | 0^{m-1}) \& B[\alpha].$$

c’est-à-dire que D est tout d’abord décalé d’une position sur la gauche. Ensuite, parmi les états potentiellement valide(s) nous rajoutons le premier par l’opération OR ($| 0^{m-1}$) qui

correspond à la boucle étiquetée par Σ sur l'état initial de l'automate non déterministe. Enfin, de tous ces états, nous ne gardons que ceux qui correspondent à une occurrence de α dans le motif, grâce à l'opération $(\& B[\alpha])$.

Après ce calcul, pour tester si nous avons trouvé une occurrence du texte, il suffit de vérifier si l'état final de l'automate est actif, donc s'il existe un 1 à la position la plus à gauche. Nous utilisons donc le test $(D \& 10^{m-1} \neq 0^m)$.

Le pseudo-code complet de l'algorithme Shift-And est illustré figure 14.

Shift-And ($p = p_1p_2 \dots p_m, T = t_1t_2 \dots t_n$)

1. **Précalcul**
2. **Pour** $c \in \Sigma$ **Faire** $B[c] \leftarrow 0^m$
3. **Pour** $j \in 1 \dots m$ **Faire** $B[p_j] \leftarrow B[p_j] | 0^{m-j}10^{j-1}$
4. **Recherche**
5. $D \leftarrow 0^m$
6. **Pour** $pos \in 1 \dots n$ **Faire**
7. $D \leftarrow ((D \ll 1) | 0^{m-1}1) \& B[t_{pos}]$
8. **Si** $D \& 10^{m-1} \neq 0^m$ **Faire** report d'une occurrence en $pos - m + 1$
9. **Fin du Pour**

FIG. 14 – Algorithme **Shift-And**.

Le précalcul du Shift-And demande $O(|\Sigma| + m)$ opérations et $|\Sigma|$ mémoire. La recherche se fait en $O(n)$.

L'algorithme Shift-Or (qui est plus connu que le Shift-And) est en fait une petite optimisation de l'algorithme Shift-And qui permet d'éviter le OR du calcul au cœur de l'algorithme. Nous remplaçons

$$D \leftarrow ((D \ll 1) | 0^{m-1}1) \& B[t_{pos}]$$

Par

$$D_1 \leftarrow (D_1 \ll 1) | B_1[t_{pos}]$$

Pour cela il faut que le tableau B_1 soit en fait le complémentaire de B , que D_1 soit initialisé à 1 partout, et enfin que le test pour savoir si le dernier état est valide soit $D_1 \& 10^{m-1} = 0^m$. Cet algorithme est un peu plus rapide que le Shift-And, plus astucieux mais moins lisible !

Exemples Voici deux exemples complets du fonctionnement du Shift-And, le premier sur un texte en anglais, le deuxième sur une séquence d'ADN.

Nous recherchons le mot “**announce**” dans le texte “**annual_announce**”. L'automate non déterministe de recherche de ce motif est montré sur la figure 15.

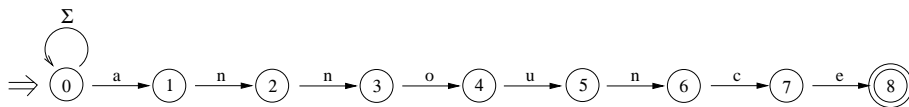


FIG. 15 – Automate non déterministe qui reconnaît tous les préfixes du mot “**announce**”.

$$B = \begin{cases} \text{a} & 00000001 \\ \text{c} & 01000000 \\ \text{e} & 10000000 \\ \text{n} & 00100110 \\ \text{o} & 00001000 \\ \text{u} & 00010000 \\ * & 00000000 \end{cases}$$

- $D = 00000000$
 $(D \ll 1) | 1 = 00000001$
- 1.. $\frac{\text{Lect. de a } 00000001}{D = 00000001}$
 - 2.. $\frac{\text{Lect. de n } 00100110}{D = 00000010}$
 - 3.. $\frac{\text{Lect. de n } 00100110}{D = 00000101}$
 - 4.. $\frac{\text{Lect. de u } 00010000}{D = 00000000}$
 - 5.. $\frac{\text{Lect. de a } 00000001}{D = 00000001}$
 - 6.. $\frac{\text{Lect. de l } 00000000}{D = 00000000}$
 - 7.. $\frac{\text{Lect. de - } 00000000}{D = 00000000}$

- 8.. $\frac{\text{Lect. de a } 00000001}{D = 00000001}$
- 9.. $\frac{\text{Lect. de n } 00100110}{D = 00000010}$
- 10.. $\frac{\text{Lect. de n } 00100110}{D = 00000101}$
- 11.. $\frac{\text{Lect. de o } 10001000}{D = 00001001}$
- 12.. $\frac{\text{Lect. de u } 00010000}{D = 00010001}$
- 13.. $\frac{\text{Lect. de n } 00100110}{D = 00100001}$
- 14.. $\frac{\text{Lect. de c } 01000000}{D = 01000001}$
- 15.. $\frac{\text{Lect. de e } 10000000}{D = 10000001}$

The last bit is set, we mark an occurrence.

Exemple sur une séquence d'ADN. Nous recherchons ATATA dans la séquence AGATAC-GATATATAC.

$$B = \begin{cases} \text{A} & 10101 \\ \text{T} & 01010 \\ * & 00000 \end{cases}$$

- $D = 00000$
- 1.. $\frac{\text{Lect. de A } 10101}{D = 00001}$
 - 2.. $\frac{\text{Lect. de G } 00000}{D = 00000}$
 - 3.. $\frac{\text{Lect. de A } 10101}{D = 00001}$

- 4.. $\frac{\text{Lect. de T } 01010}{D = 00010}$
- 5.. $\frac{\text{Lect. de A } 10101}{D = 00101}$
- 6.. $\frac{\text{Lect. de C } 00000}{D = 01011}$
- 7.. $\frac{\text{Lect. de G } 00000}{D = 00001}$
- 8.. $\frac{\text{Lect. de A } 10101}{D = 00001}$

- 9.. $\frac{\text{Lect. de T } 01010}{D = 00010}$
- 10.. $\frac{\text{Lect. de A } 10101}{D = 00101}$
- 11.. $\frac{\text{Lect. de T } 01010}{D = 01011}$
- 12.. $\frac{\text{Lect. de A } 10101}{D = 10101}$

The last bit is set, we mark an occurrence.

$$\begin{array}{r}
 \phantom{\text{Lect. de T}} \\
13.. \text{Lect. de T} \\
\hline
 \phantom{\text{Lect. de T}} \\
D =
\end{array}
\qquad
\begin{array}{r}
 \phantom{\text{Lect. de A}} \\
14.. \text{Lect. de A} \\
\hline
 \phantom{\text{Lect. de A}} \\
D = \\
\text{Le dernier bit vaut 1, nous} \\
\text{reportons une occurrence.}
\end{array}
\qquad
\begin{array}{r}
 \phantom{\text{Lect. de C}} \\
15.. \text{Lect. de C} \\
\hline
 \phantom{\text{Lect. de C}} \\
D =
\end{array}$$

Résumé de la section La recherche avant par automate permet les complexités suivantes :

$$\begin{array}{l}
\text{précalcul} \\
\text{recherche :}
\end{array}
\left| \begin{array}{l}
O(m) \\
O(n) \text{ dans le pire des cas} \\
O(n) \text{ en moyenne.}
\end{array} \right.$$

Par rapport à l’algorithme naïf du début de cette section, après tous ces efforts nous avons finalement juste.. amélioré la complexité dans le pire des cas !

2.3 Comment faire mieux ? La recherche arrière par facteurs

Dans le pire des cas, il est impossible de faire mieux que $O(n)$. Mais en moyenne (dans un modèle de Bernouilli équiprobable), une borne inférieure en nombre de comparaisons en $O(\frac{n \log m}{m})$ a été prouvée. C’est une complexité sous-linéaire, c’est-à-dire qu’en moyenne il n’est pas nécessaire de lire tous les caractères du texte pour trouver le motif recherché !

Comment atteindre cette borne avec un algorithme efficace en pratique ? Une idée récente (1993) et puissante est la suivante.

Nous déplaçons une fenêtre, dite fenêtre active, de la taille du motif sur le texte, comme sur la figure 16. Dans cette fenêtre, nous recherchons de la fin de la fenêtre vers son début un facteur du motif. Si la reconnaissance de facteur échoue sur un caractère σ , alors σu n’est pas un facteur du texte et la fenêtre active peut être déplacée sans risque après σ . Si la reconnaissance réussit, le seul facteur de la taille du motif est le motif lui-même et une occurrence a été identifiée.

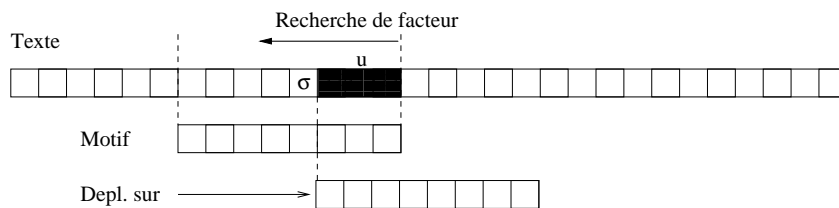


FIG. 16 – Idée de base pour déplacer la fenêtre active en utilisant l’approche par facteurs. Si la reconnaissance de facteur échoue sur σ , alors σu n’est pas un facteur du texte et la fenêtre active peut être déplacée sans risque après σ .

Cette approche mène à un algorithme optimal en moyenne, à condition d’être capable de reconnaître efficacement un facteur d’un motif donné. Prouvons le !

Pour analyser la complexité moyenne d’un algorithme ”générique” qui utilise cette approche, supposons que nous savons reconnaître un facteur u d’un motif p en $O(|u|)$. L’algo-

rithme est le suivant. nous déplaçons une fenêtre de recherche de la taille du motif de gauche à droite sur le texte. Pour chaque nouvelle position de cette fenêtre nous lisons à l'envers à partir de sa fin sans le texte un mot u de taille $3 \log_{|\Sigma|}(m)$. Si ce mot n'est pas un facteur du texte, nous déplaçons la fenêtre de lecture juste après le début de ce mot. Sinon, nous appliquons l'algorithme naïf de recherche avant de gauche à droite jusqu'à déplacer la fenêtre d'au moins $m/2$ caractères sur la droite. Plaçons nous dans le cas où $3 \log_{|\Sigma|}(m) < m/2$, ce qui est vrai dès que le motif recherché est assez suffisamment grand. Estimons maintenant le nombre d'opérations qu'il nous faut en moyenne pour déplacer la fenêtre de recherche d'au moins $m/2$ caractères sur la droite.

Il y a au maximum m facteurs dans le motif de taille m , un par position, donc la probabilité que u soit un facteur du texte est majorée par $\frac{m}{|\Sigma|^{3 \log_{|\Sigma|}(m)}}$, c'est-à-dire le nombre facteurs acceptables dans le motif divisé par le nombre de mots possible de taille $3 \log_{|\Sigma|}(m)$ sur un alphabet Σ . Notons que $m/|\Sigma|^{3 \log_{|\Sigma|}(m)} = m/m^3 = 1/m^2$. Dans cette situation, déplacer la fenêtre de $m/2$ caractères sur la droite par l'algorithme naïf nous coûte dans le pire des cas $(m/2)m = m^2/2$ comparaisons. Si u n'est pas un facteur du texte, donc avec une probabilité minorée de $\left(1 - \frac{m}{|\Sigma|^{3 \log_{|\Sigma|}(m)}}\right)$, le déplacement d'au moins m^2 caractères ne nous coûte que $3 \log_{|\Sigma|}(m)$, c'est-à-dire la lecture du mot u lui même. Pour lire le texte en entier, nous devons déplacer la fenêtre au maximum $2n/m$ fois de $m/2$ caractères. Nous obtenons une borne supérieure sur la complexité moyenne de :

$$\frac{2n}{m} \left(3 \log_{|\Sigma|}(m) \left(1 - \frac{m}{|\Sigma|^{3 \log_{|\Sigma|}(m)}} \right) + \frac{m^2}{2} \left(\frac{m}{|\Sigma|^{3 \log_{|\Sigma|}(m)}} \right) \right)$$

que nous pouvons simplifier en

$$\frac{2n}{m} \left(3 \log_{|\Sigma|}(m) \left(1 - \frac{1}{m^2} \right) + \frac{m^2}{2} \left(\frac{1}{m^2} \right) \right) = \frac{6n \log_{|\Sigma|}(m)}{m} - \frac{6n \log_{|\Sigma|}(m)}{m^3} + \frac{n}{m}$$

ce qui peut être facilement borné à son tour par :

$$O \left(\frac{n \log_{|\Sigma|}(m)}{m} \right).$$

cette approche est donc optimale en moyenne !

Dans la suite de cette section nous allons légèrement modifier cet algorithme générique pour en construire un efficace en pratique.

2.3.1 Approche BDM

Dans le cas général, nous rencontrons un problème dit *d'indexation*, que nous aborderons à la section 4. Il existe plusieurs structures pour rechercher un facteur d'un texte, dont l'arbre des suffixes, la table des suffixes, l'automate des suffixes et bien d'autres. L'algorithme Backward Dawg Matching (BDM) [16] a été originellement présenté avec un automate des suffixes pour la raison qui suit.

Comme pour l'approche par recherche avant, imaginons un automate non déterministe qui reconnaisse tous les facteurs du motif lus de droite à gauche. La figure 17 montre un tel automate pour le motif **ananas**.

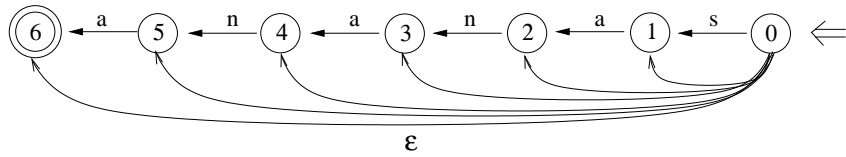


FIG. 17 – Automate non déterministe reconnaissant les facteurs du motif **ananas** lus de droite à gauche.

Tout comme dans le cas de la recherche avant, à partir de cet automate, deux grandes familles de solutions s’offrent à nous : soit en simulant ou construisant une version déterministe, soit faisant fonctionner l’automate non déterministe en utilisant le parallélisme de registre.

Il est possible de construire l’automate déterministe correspondant en temps linéaire : c’est *l’automate des suffixes*. L’automate des suffixes du mot **ananas** lu à l’envers est représenté sur la figure 18. Nous ne verrons pas cette construction dans ce cours, vous pouvez la trouver dans [4]-p182. Intéressons nous plutôt à la version utilisant le parallélisme de registre.

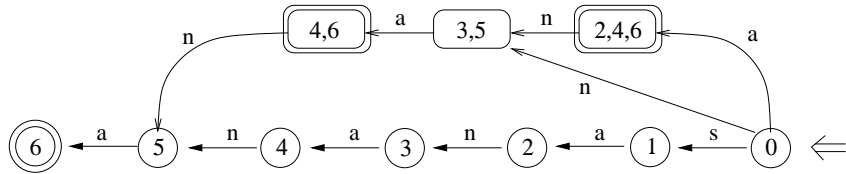


FIG. 18 – Automate déterministe, dit *des suffixes*, reconnaissant les facteurs du motif **ananas** lus de droite à gauche.

L’approche BDM est souvent affinée de la manière qui suit. Nous sommes en mesure lors de la lecture des caractères de la fenêtre de la droite vers la gauche de reconnaître les suffixes du motif lu à l’envers, donc en fait ses préfixes ! Si nous conservons la position dans la fenêtre du dernier préfixe rencontré, nous pouvons décaler ensuite le début de la fenêtre recherche à cette position. Cette idée est montrée sur la figure 19 dans laquelle le plus grand préfixe rencontré est sauvegardé dans la variable *last*.

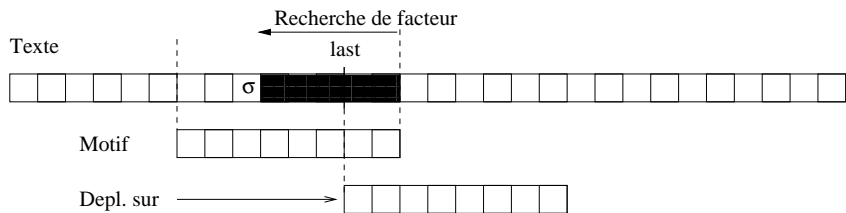


FIG. 19 – Affinement de l’idée de base pour déplacer la fenêtre active en utilisant l’automate des suffixes. La fenêtre active peut être déplacée sans risque à la position *last* qui enregistre le plus long préfixe rencontré non égal au motif lui-même.

2.3.2 BNDM

L’algorithme BNDM (pour Backward Nondeterministic Dawg Matching) [15, 16] est le pendant du Shift-And et du Shift-Or en recherchant cette fois le motif suivant l’approche

DBM. Nous utilisons une simulation de l'automate non déterministe des suffixes faite grâce aux opérations sur les registres du processeur. Le pseudo-code du BNDM est montré figure 20.

Précalcul. Le précalcul est le même que pour le Shift-And, excepté que le mot est lu dans le sens contraire.

Recherche. Pour chaque nouvelle position de la fenêtre, l'ensemble des états actifs de l'automate est enregistré dans un registre D initialisé à 1^m car chaque état est actif au début (ce qui correspond aux ε -transitions de la figure 17). Ensuite le texte est lu de droite à gauche de la fin de la fenêtre courante au travers de l'automate que nous simulons. Le registre D est mis à jour simplement avec un décalage et un ET. Pour des raisons d'entrelacement des tests, ils ne sont pas fait à la même ligne. Le décalage est fait ligne 16 alors que le ET est fait ligne 10. Le test ligne 12 permet de savoir si nous avons lu un préfixe du motif. Si c'est le cas, soit ce préfixe n'est pas égal au motif lui-même et dans ce cas nous sauvegardons simplement sa taille dans la variable $last$ ligne 13, soit nous avons trouvé une occurrence marquée ligne 14. la lecture des caractères dans la fenêtre se répète jusqu'à ce qu'il n'y ait plus d'état valide, ce qui est le test d'entrée de la boucle **Tant que** ligne 6. A la sortie de cette boucle, que le motif ait été trouvé ou non, nous décalons la fenêtre de recherche jusqu'à $last$ et recommençons la boucle générale ligne 6 si nécessaire.

```

BNDM ( $p = p_1p_2 \dots p_m$ ,  $T = t_1t_2 \dots t_n$ )
1.  Précalcul
2.    Pour  $c \in \Sigma$  Faire  $B[c] \leftarrow 0^m$ 
3.    Pour  $j \in 1 \dots m$  Faire  $B[p_j] \leftarrow B[p_j] \mid 0^{j-1}10^{m-j}$ 
4.  Recherche
5.     $pos \leftarrow 0$ 
6.    Tant que  $pos \leq n - m$  Faire
7.       $j \leftarrow m$ ,  $last \leftarrow m$ 
8.       $D \leftarrow 1^m$ 
9.      Tant que  $D \neq 0^m$  Faire
10.         $D \leftarrow D \& B[t_{pos+j}]$ 
11.         $j \leftarrow j - 1$ 
12.        Si  $D \& 10^{m-1} \neq 0^m$  Faire
13.          Si  $j > 0$  Faire  $last \leftarrow j$ 
14.          Sinon marquer une occurrence en  $pos + 1$ 
15.        Fin du Si
16.         $D \leftarrow D \ll 1$ 
17.      Fin du Tant que
18.       $pos \leftarrow pos + last$ 
19.    Fin du Tant que

```

FIG. 20 – Pseudo-code du **BNDM**.

Le code en langage C du BNDM est accessible à l'adresse <http://www.liafa.jussieu.fr/~raffinot/bndm.html>. Voici un exemple de fonctionnement du BNDM sur une séquence d'ADN. Nous recherchons le motif ATATA dans la séquence AGATACGATATATAC.

aussi assez courant de supprimer des parenthèses en définissant un ordre de préférence sur les opérateurs ‘*’, ‘.’, ‘|’, mais pour simplifier nous ne le ferons pas. Les symboles ‘.’, ‘|’, ‘*’ sont les *opérateurs*.

Définition 2.3 *Le langage $L(RE)$ représenté par une expression régulière RE est un ensemble de mots construits sur l’alphabet Σ , qui est lui aussi défini récursivement sur la structure de l’expression régulière RE de la façon suivante :*

- si RE est ε , alors $L(RE) = \{\varepsilon\}$, le mot vide.
- si RE est $\alpha \in \Sigma$, alors $L(RE) = \{\alpha\}$, un mot composé d’un simple caractère.
- si RE est de la forme (RE_1) , alors $L(RE) = L(RE_1)$.
- si RE est de la forme $(RE_1 \cdot RE_2)$, alors $L(RE) = L(RE_1) \cdot L(RE_2)$, où $W_1 \cdot W_2$ est l’ensemble de mots w tel que $w = w_1w_2$, avec $w_1 \in W_1$ et $w_2 \in W_2$. L’opérateur ‘.’ représente en fait l’opération classique de concaténation sur les mots.
- si RE est de la forme $(RE_1 | RE_2)$, alors $L(RE) = L(RE_1) \cup L(RE_2)$, est l’union des deux langages.
- si RE est (RE_1^*) , alors $L(RE) = L(RE_1)^* = \bigcup_{i \geq 0} L(RE_1)^i$, où $L^0 = \{\varepsilon\}$ et $L^i = L \cdot L^{i-1}$ pour tout L . C’est-à-dire, le résultat est l’ensemble des mots formés par concaténation de mots de $L(RE_1)$. L’opérateur ‘*’ est appelé ‘étoile’ ou encore ‘opérateur de fermeture de Kleene’.

Par exemple, $L((AT|GA)((AG|AAA)^*)) = \{ AT, GA, ATAG, GAAG, ATAAA, GAAAA, ATAGAG, ATAGAAA, ATAAAAG, ATAAAAA, GAAGAG, GAAGAAA, \dots \}$. Remarquez que, selon la définition de l’opérateur étoile, Σ^* est l’ensemble des mots que l’on peut construire sur l’alphabet Σ .

La *taille* d’une expression régulière RE est le nombre de caractères de Σ qu’elle contient. La taille de $(AT|GA)((AG|AAA)^*)$ est 9. Les complexités mesurées par la suite se basent sur cette mesure.

2.4.1 Parsing en arbre binaire

Le terme ‘Parsing’ signifie la traduction en structure de donnée informatique de la séquence de symboles qui constituent l’expression régulière (pouvant contenir des erreurs de formation) que l’utilisateur entre au programme. Dans notre cas, il existe plusieurs type de données qui peuvent être utilisées, mais la plus simple et la plus naturelle est un arbre binaire.

Un arbre binaire est soit (a) un arbre vide noté Θ , soit (b) un nœud qui possède deux fils, un gauche et un droit. A chaque nœud non vide de l’arbre est associé une étiquette, dans notre cas ce sera soit un caractère de l’alphabet, soit un opérateur. Pour mieux comprendre l’idée, la figure 21 permet de visualiser l’expression $(AT|GA)((AG|AAA)^*)$ sous forme d’arbre. Les parenthèses de l’expression ‘plate’ correspondent à des niveaux dans l’arbre.

Passer d’une expression régulière ‘plate’ à un arbre n’est pas si facile. C’est en fait le début de la théorie des langages et vous pouvez trouver de nombreuses références sur le sujet [1, 6]. Nous simplifions cette partie en utilisant la fonction **Parse** de la figure 22. Elle prend en entrée une expression plate sous forme d’une chaîne de caractères $p = p_1p_2 \dots p_m$ dont on assure qu’elle se termine par un symbole spécial ‘\$’. Elle prend aussi en paramètre la position *last* du dernier élément de p qui a déjà été traité.

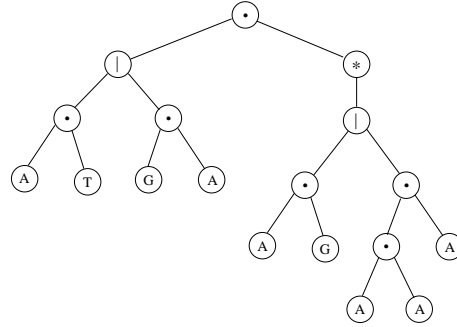


FIG. 21 – Représentation sous forme d’arbre de l’expression $(AT|GA)((AG|AAA)^*)$. Les noeuds vides Θ n’apparaissent pas sur la figure.

2.4.2 Stratégies de recherche exacte

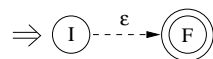
Il existe de nombreuses stratégies pour rechercher une expression régulière dans un texte. La figure 23 résume les plus classiques.

Nous nous focalisons dans ce document sur la stratégie marquée en gras sur la figure 23, c’est-à-dire que nous commençons par construire un automate de non déterministe (dit de *Thompson*) que nous utilisons directement pour effectuer la recherche dans le texte. La taille de l’automate déterministe sera proportionnelle à celle de l’expression régulière, en $O(m)$ et le temps de recherche en $O(nm)$.

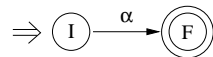
2.4.3 Automate de Thompson

L’automate de Thompson est un automate non déterministe qui est construit récursivement sur la forme de l’expression régulière suivant les cinq points que voici :

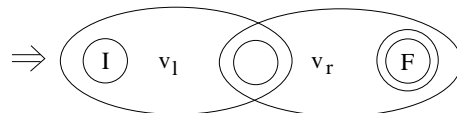
- (i) Construction pour le mot vide. L’automate est simplement fait de deux états liés ensemble par une ε -transition.



- (ii) Pour un simple caractère α , la construction est similaire sauf que la transition est étiquetée par α .



- (iii) Construction pour l’opérateur de concaténation \square (v_l, v_r). Les deux automates de Thompson des deux fils sont reliés entre eux par une ε -transition.



- (iv) La construction pour un nœud d’union $[|]$ (v_l, v_r) utilise des ε -transitions. L’idée est de transcrire le fait que nous pouvons entrer dans les deux automates des deux fils. On ajoute alors deux états, un initial I et un final J . I est relié par des ε -transitions aux états initiaux des automates de Thompson des deux fils. Et les états finaux de ces deux

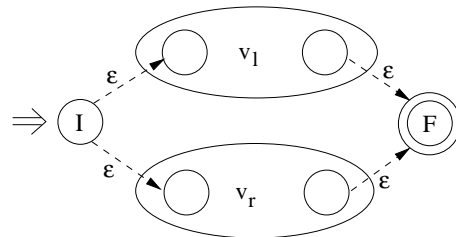
```

Parse( $p = p_1 p_2 \dots p_m, last$ )
1.  $v \leftarrow \theta$ 
2. Tant que  $p_{last} \neq \$$  Faire
3.   Si  $p_{last} \in \Sigma$  OU  $p_{last} = \varepsilon$  Faire /* caractère normal */
4.      $v_r \leftarrow$  Créer un nœud avec  $p_{last}$ 
5.     Si  $v \neq \theta$  Faire  $v \leftarrow \boxed{\cdot}$ ( $v, v_r$ )
6.     Sinon  $v \leftarrow v_r$ 
7.      $last \leftarrow last + 1$ 
8.   Sinon Si  $p_{last} = '|'$  Faire /* opérateur d'union */
9.      $(v_r, last) \leftarrow$  Parse( $p, last + 1$ )
10.     $v \leftarrow [ | ]$ ( $v, v_r$ )
11.   Sinon Si  $p_{last} = '*'$  Faire /* opérateur étoile */
12.     $v \leftarrow \boxed{*}$ ( $v$ )
13.     $last \leftarrow last + 1$ 
14.   Sinon Si  $p_{last} = '('$  Faire /* parenthèse ouvrante */
15.     $(v_r, last) \leftarrow$  Parse( $p, last + 1$ )
16.     $last \leftarrow last + 1$ 
17.    Si  $v \neq \theta$  Faire  $v \leftarrow \boxed{\cdot}$ ( $v, v_r$ )
18.    Sinon  $v \leftarrow v_r$ 
19.   Sinon Si  $p_{last} = ')'$  Faire /* parenthèse fermante */
20.    Retourner ( $v, last$ )
21.   Fin du Si
22. Fin du Tant que
23. Retourner ( $v, last$ )

```

FIG. 22 – Procédure récursive pour transformer une expression “plate” en un arbre. θ est l’arbre vide.

automates sont eux reliés à J , toujours par des ε -transitions. Un chemin de I à J doit passer par un des deux automates.



(v) La construction pour un nœud étoile $\boxed{*}$ (v) est basée sur le même principe. L’idée est simplement de pouvoir revenir de l’état final de l’automate de Thompson du fils v vers son état initial pour pouvoir répéter autant de fois que l’on veut cette partie. Comme l’étoile permet aussi de ne pas passer par l’automate et de reconnaître simplement le mot vide, on ajoute deux états I et J qui permettent, soit de passer par l’automate de v , soit de passer directement de I à J par une ε -transition.

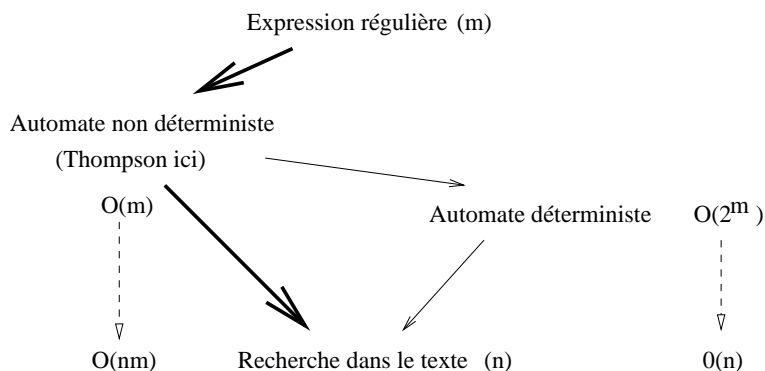
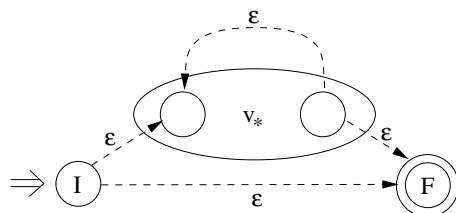


FIG. 23 – Stratégies classiques de recherche d’une expression régulière.



L’automate de Thompson complet est construit par un parcours récursif de l’arbre de l’expression régulière.

Exemple de construction d’un automate de Thompson Nous construisons pas à pas l’automate de Thompson de $(AT|GA)((AG|AAA)^*)$ à partir de sa représentation en arbre donnée figure 21. La construction est montrée sur la figure 2.4.3.

2.4.4 Recherche avec l’automate de Thompson

L’automate de Thompson de l’expression régulière RE reconnaît le langage $L(RE)$ de cette expression. Pour faire la recherche dans une séquence, nous avons besoin de reconnaître $\Sigma^*L(RE)$, c’est-à-dire n’importe quel mot construit sur l’alphabet Σ suivi d’un mot reconnu par RE . Comme pour la recherche d’un mot, l’automate le plus simple qui correspond est celui de RE auquel nous ajoutons une boucle qui reconnaît tous les caractères. Par exemple, pour l’expression $A(T^*)$, l’automate, dit alors *de recherche*, est montré sur la figure 25.

La recherche se fait en avant, en lisant les caractères les uns après les autres et en actualisant la liste des états actifs. La figure 26 montre les états actifs de l’automate de recherche de $A(T^*)$ lors de la lecture de $ACATTTG$.

2.4.5 Sur le report des occurrences

La recherche d’une expression régulière se conçoit bien avec un automate. Cependant, les occurrences résultantes peuvent se superposer, être imbriquées les unes dans les autres et il est difficile en bioinformatique de clarifier exactement les occurrences que nous voulons obtenir et de modifier l’approche par automate pour n’obtenir que ces dernières. Ces techniques sortent du cadre de ce cours et les plus classiques sont détaillées dans [14].

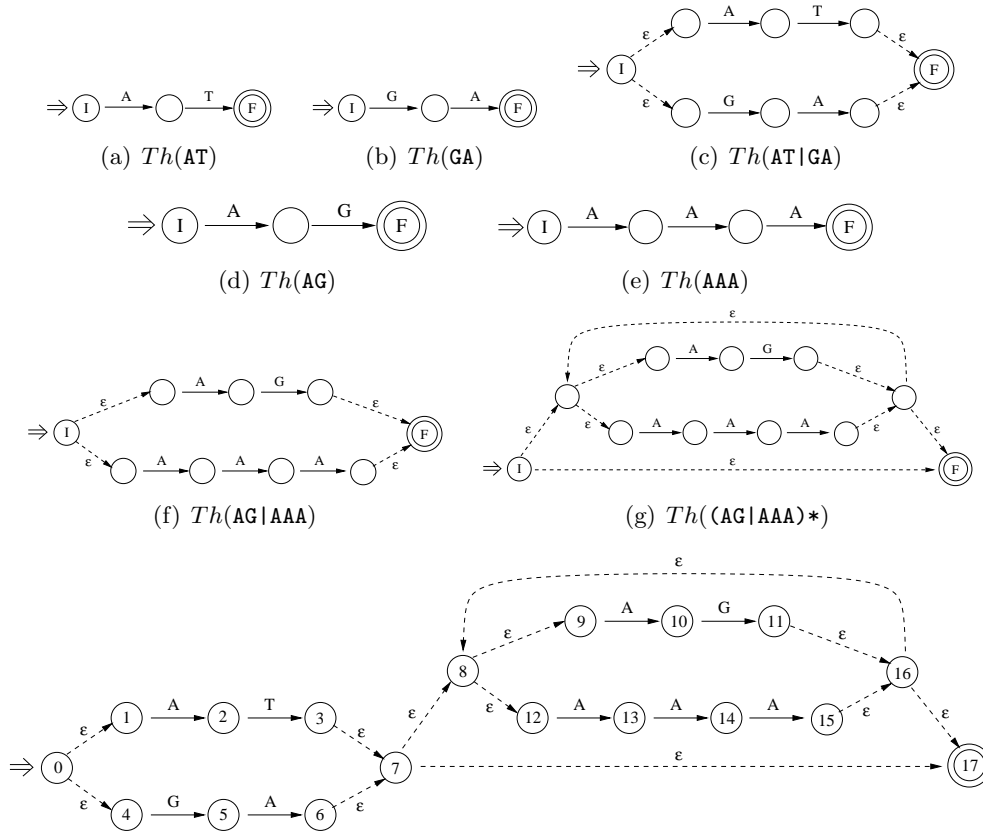


FIG. 24 – Construction de l’automate de Thompson pour l’expression régulière $((AA|AT)((AG|AAA)^*)$.

3 Recherche approchée

Nous avons vu différentes techniques de recherche exacte d’un motif dans un texte, c’est-à-dire que nous voulons trouver exactement le motif dans le texte. Cependant, en bioinformatique principalement, cette notion est souvent trop forte à cause des erreurs de séquençage, des SNP et du polymorphisme. Nous devons donc faire des recherches “approchées”, c’est-à-dire que nous cherchons un sous-mot du texte ou de la séquence qui soit “pas trop loin” du motif. Encore faut-il définir correctement ce que veut dire pas trop loin, et que cela ait un sens biologique !

Historiquement, de nombreuses notions de recherches approchées sont apparues, la première et l’inspiration de beaucoup d’autres étant basée sur la distance entre mots de Levenshtein.

3.1 Distance de Levenshtein

La distance de Levenshtein [11] est le plus petit nombre d’erreurs pour transformer un mot $X = x_1 \dots x_m$ en un mot $Y = y_1 \dots y_n$. Les erreurs peuvent être de trois ordres : (a) insertion d’une lettre dans le mot x , (b) déletion d’une lettre dans le mot x et (c) substitution d’une lettre par une autre.

Pour la calculer, nous utilisons une méthode dite de “programmation dynamique” dans

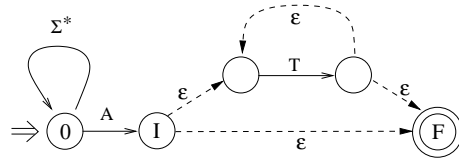


FIG. 25 – Automate de recherche de l'expression $A(T^*)$. La boucle initiale permet de reconnaître le langage $\Sigma^*L(A(T^*))$.

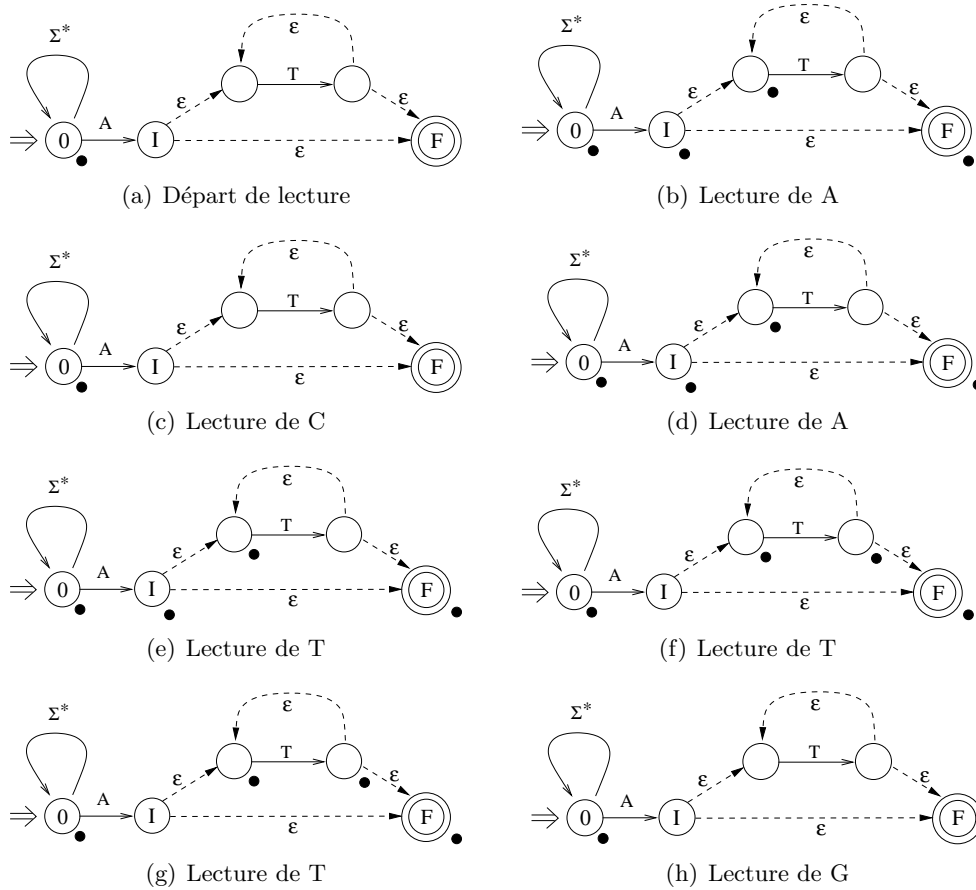


FIG. 26 – Exemple de lecture du texte ACATTTG par l'automate de Thompson de l'expression $A(T^*)$. Les états actifs sont marqués par un rond noir.

une table T de taille $(m + 1) * (n + 1)$ colonne par colonne en utilisant la formule suivante :

$$T_{0,0} = 0, T_{0,j} = j, T_{i,0} = i$$

$$T_{i,j} = \min \begin{cases} T_{i-1,j-1} + \delta(x_i, y_j) \\ T_{i,j-1} + 1 \\ T_{i-1,j} + 1 \end{cases}$$

où $\delta(a, b) = 1$ si $a = b$ et 0 sinon. La valeur de la distance est celle de la cellule $T_{n,m}$.

Pour prouver qu'avec cette approche nous calculons bien la distance d'édition l'observation suivant est importante : les insertions, délétions et insertions de lettres correspondant à la distance d'édition peuvent se faire dans n'importe quel ordre. Considérons alors l'opération faite sur le dernier caractère x_m de X contre le dernier caractère $y - n$ de Y . La valeur de la distance de X contre Y est le minimum de

- la distance de $x_1 \dots x_{m-1}$ contre Y en ajoutant une erreur pour la délétion du caractère x_m ;
- la distance de $x_1 \dots x_m$ contre $y_1 \dots y_{n-1}$ en ajoutant une erreur pour l'insertion du caractère x_m ;
- la distance de $x_1 \dots x_{m-1}$ contre $y_1 \dots y_{n-1}$ en ajoutant une erreur si $x_m \neq y_n$.

En développant cette approche nous obtenons la formule précédente. Voici un exemple de calcul de distance entre *natif* et *animation*.

		<i>a</i>	<i>n</i>	<i>i</i>	<i>m</i>	<i>a</i>	<i>t</i>	<i>i</i>	<i>o</i>	<i>n</i>
	0	1	2	3	4	5	6	7	8	9
<i>n</i>	1	1	1	2	3	4	5	6	7	8
<i>a</i>	2	1	2	2	3	3	4	5	6	7
<i>t</i>	3	2	2	3	3	4	3	4	5	6
<i>i</i>	4	3	3	2	3	4	4	3	4	5
<i>f</i>	5	4	4	3	3	4	5	4	4	5

Et entre *naturel* et *manuel*.

		<i>m</i>	<i>a</i>	<i>n</i>	<i>u</i>	<i>e</i>	<i>l</i>
	0	1	2	3	4	5	6
<i>n</i>	1	1	2	2	3	4	5
<i>a</i>	2	2	1	2	3	4	5
<i>t</i>	3	3	2	2	3	4	5
<i>u</i>	4	4	3	3	2	3	4
<i>r</i>	5	5	4	4	3	3	4
<i>e</i>	6	6	5	5	4	3	4
<i>l</i>	7	7	6	6	5	4	3

Le calcul de la distance d'édition est en $O(mn)$ en temps mais peut être fait en $O(m)$ espace.

Une fois la distance minimale calculée, il est souvent nécessaire de connaître la liste des opérations à effectuer pour transformer X en Y . Mais en fait ce n'est pas une liste d'opérations, c'est un ensemble de liste(s), chacune correspondant à un chemin, dit *optimal*, différent dans la matrice. Par exemple, sur la matrice 27, 3 chemins différents mènent à la distance minimale.

Lorsque nous écrivons un des chemins explicitement, nous réalisons ce qui est souvent appelé un *alignement*. Voici les 3 alignements correspondant aux chemins optimaux de la figure

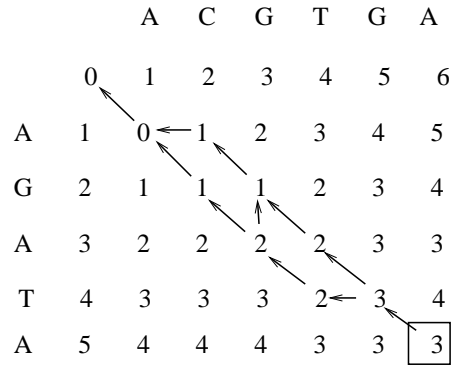


FIG. 27 – Chemins optimaux de AGATA contre ACGTGA

27.

A	C	G	T	G	A	A	C	G	-	T	G	A	A	C	G	T	G	A
A	-	G	A	T	A	A	-	G	A	T	-	A	A	G	A	T	-	A

La distance de Levenshtein de X contre Y et la recherche de chemins optimaux correspond en bioinformatique à un alignement dit “global”. Mais dans beaucoup de cas il est plutôt nécessaire de rechercher de manière approchée un mot dans un texte ou mee de comparer des facteurs de X contre des facteurs de Y .

3.2 Recherche avec la distance de Levenshtein

Pour modifier le calcul de la distance de Levensthein d’un mot X contre un mot Y en recherche approchée d’un mot X dans un texte Y il existe une modification très simple due à Sellers [19]. Pour le calcul de la distance nous initialisons $T_{0,j}$ à j car insérer un préfixe de taille j de Y coûte j insertions. La modifications consiste juste à garder cette valeur à 0 pour tous les $T_{0,j}$. Ainsi, insérer un préfixe de Y ne coûte plus rien, autrement dit, l’alignement peut commencer sans pénalité à toutes les positions de Y . Nous obtenons alors la formule suivante :

$$T_{0,0} = 0, T_{0,j} = 0, T_{i,0} = i$$

$$T_{i,j} = \min \begin{cases} T_{i-1,j-1} + \delta(x_i, y_j) \\ T_{i,j-1} + 1 \\ T_{i-1,j} + 1 \end{cases}$$

La figure 28 montre un esemple d’une recherche approchée de AGATA dans ACGTGA-TAGAGACCG avec les alignements correspondant au minimum d’erreurs.

La figure 29 monte un nouvel exemple de recherche approchée et des alignements optimaux, cette fois-ci de GATACTGAGT dans ATGATCTCAAGTGTATA.

3.3 Needleman-Wunsch, alignement global

L’alignement global de Levenshtein a été étendu par Needleman et Wunch [17] pour utiliser un score au lieu d’erreurs, c’est-à-dire en fait pondérer plus finement chaque substitution ou

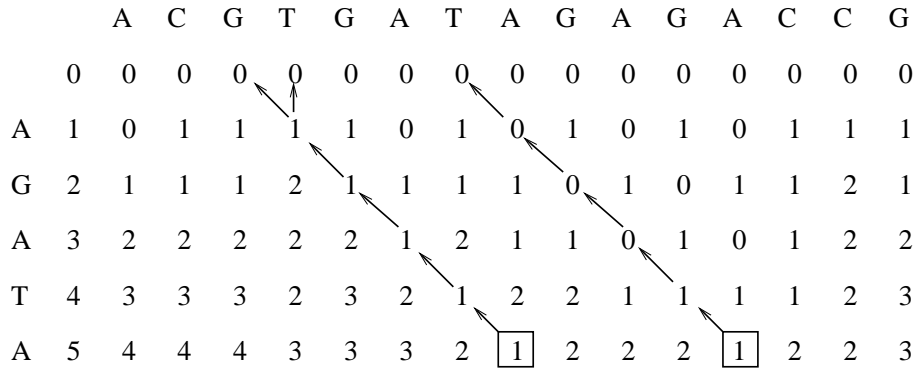


FIG. 28 – Recherche approchée de AGATA dans ACGTGATAGAGACCG

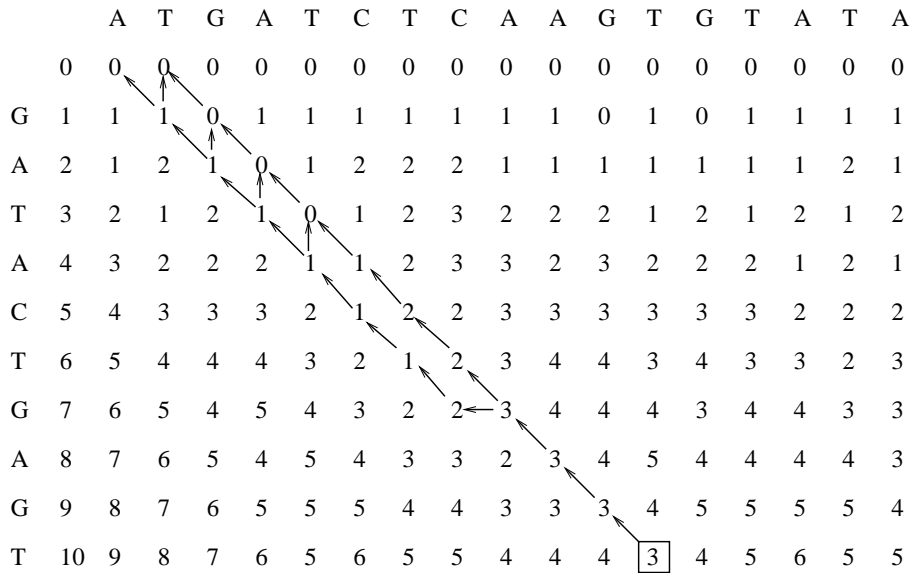


FIG. 29 – Recherche approchée de GATACTGAGT dans ATGATCTCAAGTGTATA

insertion. Dans l'approche la plus classique une matrice permet de spécifier le coût de la substitution d'un caractère par un autre, et un coût unique spécifique est associé à l'insertion (gap en anglais) quelque soit le caractère inséré et que ce soit un caractère de X ou de Y . Il existe de nombreux types de matrices de scores, certaines sont basées sur des calculs dérivants de phylogénie (Matrices PAM ou BLOSUM par exemple), d'autres sur des propriétés physico-chimiques des acides aminés, etc. Avant de faire un calcul d'alignement, il faut s'assurer que la matrice utilisée correspond bien au but de l'alignement et aux séquences alignées ! La formule générale est la suivante :

$$T_{0,0} = 0, T_{0,j} = -j * \text{insert}(y_i), T_{i,0} = -i * \text{insert}(y, i)$$

$$T_{i,j} = \max \begin{cases} T_{i-1,j-1} + \text{sub}(x_i, y_j) \\ T_{i,j-1} + \text{insert}(x_j) \\ T_{i-1,j} + \text{insert}(y_i) \end{cases}$$

Voici un exemple de matrice de score et de pondération de l'insert (ou du gap) :

	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

insert \leftarrow -5,

Avec cette matrice et ce gap, lorsque nous alignons AGATA contre ACGTGA, nous obtenons les pondérations qui suivent :

		A	C	G	T	G	A
	0	-5	-10	-15	-20	-25	-30
A	-5	10	5	0	-5	-10	-15
G	-10	5	5	14	9	4	-1
A	-15	0	4	9	10	6	14
T	-20	-5	-1	4	17	12	9
A	-25	-10	-6	-1	12	14	22

Regardons les chemins optimaux. Contrairement au cas de la distance de Levenshtein, il n'y en a cette fois plus qu'un, qui correspond à l'alignement de la figure 30, représenté de manière plus lisible comme :

$$\begin{array}{ccccccccc} & A & C & G & - & T & G & A & \\ & | & & | & & | & & | & \\ A & - & G & A & T & - & A & & \end{array}$$

3.4 Smith-Waterman, alignement local

De la même manière qu'il est possible d'étendre la distance de Levenshtein avec un score, il est possible d'étendre la recherche approchée en utilisant une matrice de score, ou même d'aligner chaque facteur de X contre chaque facteur de Y . On parle alors d'alignement *local* et d'algorithme de Smith-Waterman [20].

$$T_{0,0} = 0, T_{0,j} = 0, T_{i,0} = 0$$

		A	C	G	T	G	A	
	0	-5	-10	-15	-20	-25	-30	
A	-5	10	←5	←	0	-5	-10	-15
G	-10	5	5	14	9	4	-1	
A	-15	0	4	9	10	6	14	
T	-20	-5	-1	4	17	←12	9	
A	-25	-10	-6	-1	12	14	22	

FIG. 30 – Chemin optimal de AGATA contre ACGTGA en utilisant un score donné.

$$T_{i,j} = \max \begin{cases} 0 \\ T_{i-1,j-1} + \text{sub}(x_i, y_j) \\ T_{i,j-1} + \text{insert}(x_j) \\ T_{i-1,j} + \text{insert}(y_i) \end{cases}$$

		A	C	G	T	G	A	T	A	G	A	G	A	C	C	G
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	10	5	0	0	0	10	5	10	5	10	5	10	5	0	0
G	0	5	5	14	9	9	5	10	5	19	14	19	14	9	4	9
A	0	10	5	9	10	6	19	14	20	15	29	24	29	24	19	14
T	0	5	7	5	17	12	14	27	22	20	24	29	24	26	21	19
A	0	10	5	4	12	14	22	22	37	32	30	25	39	34	29	24

FIG. 31 – Recherche approchée de AGATA dans ACGTGATAGAGACCG

La figure 32 monte un nouvel exemple de recherche approchée et des alignements optimaux, cette fois-ci de AGATACTA dans CCCGAAACTGGG.

3.5 BLAST et FASTA

L'algorithme de Smith-Waterman est le plus utilisé pour aligner une séquence sur une autre, avec différents scores correspondant à l'étude voulue sur les séquences. Cependant, sa complexité est en temps $O(mn)$, ce qui le rend inutilisable en pratique sur de longs fragments ou simplement pour rechercher une séquence dans une importante base de séquences. Certaines heuristiques ont été développées sur un équilibre (vitesse)/(finesse de recherche). Les plus célèbres sont BLAST [2] et FASTA [18] et leurs dérivations, mais il en existe d'autres comme BLAT [7] ou plus récemment BOWTIE [10], ainsi que différents systèmes propriétaires.

	C	C	C	G	A	A	A	C	T	G	G	G
	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	10	10	10	5	0	0	0
G	0	0	0	0	9	5	7	7	5	5	9	9
A	0	0	0	0	4	19	15	17	12	7	4	6
T	0	0	0	0	0	14	15	12	14	20	15	10
A	0	0	0	0	0	10	24	25	20	15	17	12
C	0	7	7	7	2	5	19	23	32	27	22	17
T	0	2	4	4	7	2	14	18	27	40	35	30
A	0	0	1	3	2	17	12	24	22	35	37	32

FIG. 32 – Recherche approchée de AGATACTA dans CCCGAAACTGGG

4 Indexation de texte

Jusqu'ici lors de nos recherches d'un motif dans un texte, nous avons seulement considéré un précalcul sur le motif pour le rechercher le plus efficacement dans un texte donné. Mais pour rechercher un grand nombre de motifs, il peut être judicieux d'essayer de précalculer des informations sur le texte pour accélérer la recherche.

Les informations dont nous avons besoin sont principalement des positions de certains facteurs du texte. La structure de données qui va contenir ces informations de positions est souvent appelée "index".

Les indexes les plus utilisés en bioinformatique sont la table de suffixes et l'arbre des suffixes, mais il en existe d'autres, comme le DAWG ou le CDAWG que nous n'aborderons pas dans le cadre de ce cours.

4.1 Table des suffixes

Considérons l'ensemble des suffixes d'un texte donné et trions-les par ordre lexicographique (ordre du dictionnaire) croissant. Par exemple, pour le motif AGAGATGA, nous obtenons la table de la figure 33.

8	A
1	AGAGATGA
3	AGATGA
5	ATGA
7	GA
2	GAGATGA
4	GATGA
6	T

FIG. 33 – Tri par ordre lexicographique des suffixes du texte AGAGATGA

```

Borne_gauche( $P = p_1 p_2 \dots p_m, \text{TS}[0..n - 1]$ )
1.   $pos\_deb \leftarrow 0; pos\_fin \leftarrow n - 1$ 
2.  Tant que ( $pos\_fin - pos\_deb > 1$ ) Faire
3.       $pos\_mil \leftarrow (pos\_fin + pos\_deb)/2$  /* pos. de la case du milieu */
4.      Si ( $tab[pos\_mil] \geq P$ ) Faire
5.           $pos\_fin \leftarrow pos\_mil$ 
6.      Sinon
7.           $pos\_deb \leftarrow pos\_mil$ 
8.      Fin du si-sinon
9.  Fin du tant que
10. Renvoyer  $pos\_fin$ 

```

FIG. 34 – Pseudo-code de la recherche dichotomique de la borne gauche de l'intervalle correspondant à un motif P donné dans une table des suffixes TS. La recherche de la borne droite est symétrique.

La table des suffixes est simplement la permutation triée 8 1 3 5 7 2 4 6. Elle occupe classiquement $4 * n$ octets en mémoire.

4.1.1 Comment faire une recherche d'un motif sur cette permutation ?

La recherche d'un motif consiste à rechercher l'intervalle, s'il existe, de suffixes qui admettent ce motif comme préfixe. Par exemple, sur la figure 33, la recherche du motif AG doit renvoyer l'intervalle [1, 2]. Pour faire cette recherche, nous recherchons tout d'abord la borne gauche de l'intervalle, puis de manière symétrique sa borne droite. Comment rechercher une de ces bornes ?

En fait, en appliquant la méthode classique de recherche dans un tableau trié : la recherche dichotomique, en utilisant la comparaison lexicographique comme comparaison de base. Le pseudo-code de la recherche de la borne gauche de l'intervalle est donné figure 34.

Essayons l'algorithme **Borne_gauche** sur un exemple, recherchons AG dans AGAGATGA en utilisant la table des suffixes 8 1 3 5 7 2 4 6. Au début, $pos_deb \leftarrow 0$ et $pos_fin \leftarrow 7$, $pos_mil \leftarrow 3$. Comme $ATGA = \text{TS}[3] > AG$, $pos_fin \leftarrow 3$. Ensuite $pos_mil \leftarrow 3 + 0/2 = 1$. Comme $AGAGATGA = \text{TS}[1] > AG$, $pos_fin \leftarrow 1$. Et maintenant puisque $pos_fin - pos_deb = 1$, la boucle **Tant que** s'arrête et la borne gauche renvoyée est 1.

La comparaison lexicographique de P avec un élément du tableau se fait en $m = |P|$ et le nombre d'itérations maximum est $\log n$. La recherche se fait donc en temps $O(m \log n)$.

4.1.2 Comment construire cette table ?

La construction de la table des suffixes peut être faite en temps $O(n)$ en utilisant un algorithme concis mais très astucieux [8] qui dépasse le cadre de ce cours.

La table des suffixes a été proposée dans [12]. Dans cet article, pour améliorer la phase de recherche, la table des suffixes peut être complétée par une autre table de $4n$ octets qui enregistre les plus grands préfixes communs entre les couples de suffixes (Lowest Common Préfixe (LCP) en anglais) qui sont utilisés lors de toute recherche dichotomique. L'utilisation

de cette table permet d'accélérer la recherche de $O(m \log n)$ à $O(m + \log n)$. En pratique cependant la construction de cette table est compliquée et elle est rarement utilisée.

4.2 Arbre des suffixes

L'arbre des suffixes est lui aussi construit à partir de l'ensemble des suffixes, mais en les structurant sous la forme d'un arbre. Construisons un premier arbre de tous les suffixes, comme sur la figure 35-(a). Ce premier arbre est appelé *trie des suffixes*. Ce premier arbre n'est pas satisfaisant car sa taille est quadratique par rapport à la taille du texte, et une telle structure n'est plus utilisable dès que la taille du texte dépasse quelque milliers de caractères. Essayons de le compacter. Une idée simple vient à l'esprit : pourquoi ne pas supprimer les nœuds qui n'ont qu'une seule sortie, en utilisant des arêtes multicaractères ? Nous obtenons l'arbre de la figure 35-(b). Mais cela ne suffit pas pour baisser la complexité de l'occupation mémoire. Pour obtenir une structure de taille linéaire, il ne faut pas coder une transition dans l'arbre, mais la coder comme un facteur du texte dont nous pouvons indiquer pour chaque arête le début et la fin ! Nous obtenons l'arbre de la figure 36, dit "arbre des suffixes".

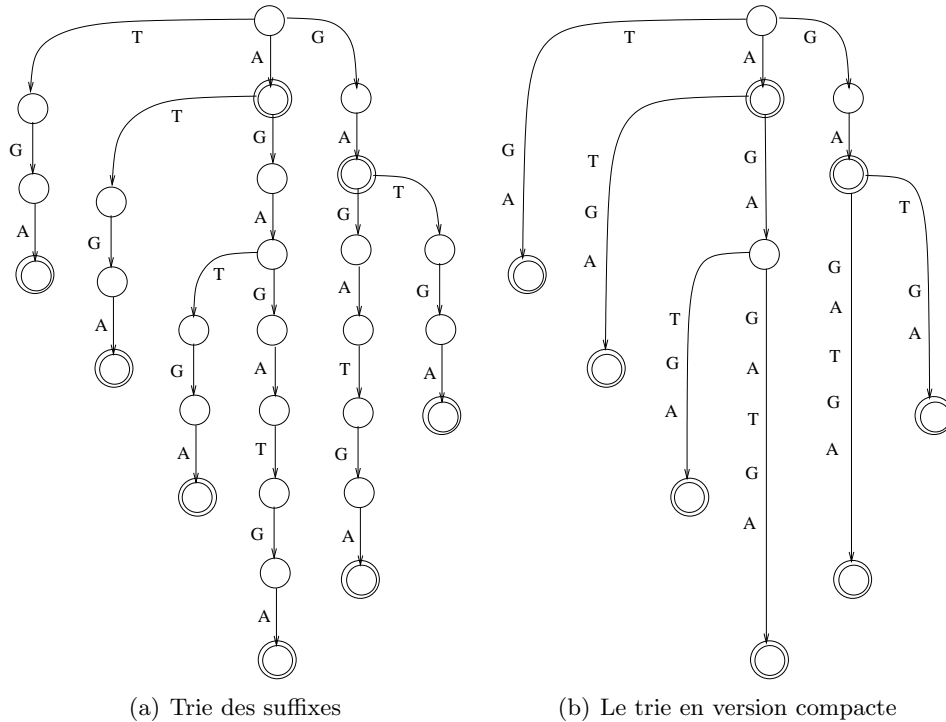


FIG. 35 – Trie et trie compacté des suffixes.

Cet arbre occupe $O(n)$ mémoire et son occupation réelle est d'environ $16n$ octets en pratique, ce qui le rend moins utilisable que la table des suffixes sur de grands textes ou séquences. Par contre, une recherche d'un motif de taille m dans cet arbre peut se faire en complexité $O(m)$ car il suffit pour chaque lettre de "descendre" dans l'arbre en suivant la bonne transition si elle existe.

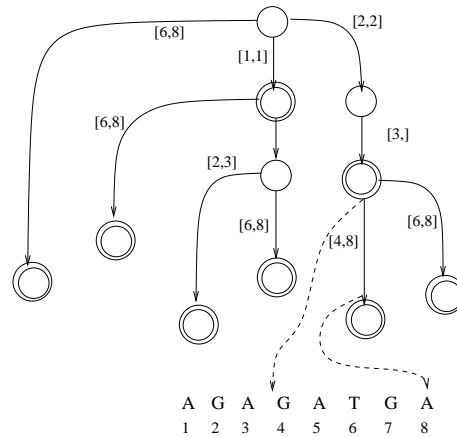


FIG. 36 – Arbre des suffixes. Sa taille est linéaire en la taille du texte.

4.2.1 Comment construire l’arbre des suffixes ?

La construction de l’arbre des suffixes dépasse elle aussi le cadre de ce cours. Elle peut se faire en temps $O(n)$ en utilisant un des multiples algorithmes linéaires de construction, dont le plus simple (mais astucieux) est celui d’Ukkonen de 1995 [21].

Références

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *J. Mol. Biol.*, 215 :403–410, 1990.
- [3] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proceedings of the 12th International Conference on Research and Development in Information Retrieval*, pages 168–175, Cambridge, MA, 1989. ACM Press.
- [4] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert, 2001.
- [5] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
- [6] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations*. Addison-Wesley, Reading, MA, 1979.
- [7] Kent W. James. Blat—the blast-like alignment tool. *Genome research*, 12(4) :656–664, April 2002.
- [8] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6) :918–936, 2006.
- [9] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1) :323–350, 1977.
- [10] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3) :R25+, 2009.

- [11] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1 :8–17, 1965.
- [12] U. Manber and G. Myers. Suffix arrays : a new method for on-line string searches. *SIAM J. Comput.*, 22(5) :935–948, 1993.
- [13] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.
- [14] E. W. Myers, P. Oliva, and K. Guimãraes. Reporting exact and approximate regular expression matches. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 91–103, Piscataway, NJ, 1998. Springer-Verlag, Berlin.
- [15] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata : Fast extended string matching. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33, Piscataway, NJ, 1998. Springer-Verlag, Berlin.
- [16] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.
- [17] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48 :443–453, 1970.
- [18] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. U.S.A.*, 85 :2444–2448, 1988.
- [19] P. H. Sellers. An algorithm for the distance between two finite sequences. *J. Comb. Theory Ser. A*, 16 :253–258, 1974.
- [20] T. F. Smith and M. S. Waterman. Identification of common molecular sequences. *J. Mol. Biol.*, 147 :195–197, 1981.
- [21] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, 1995.