L'Arbre des suffixes

Olivier Delgrange

Ce papier est destiné aux lecteurs qui ne sont pas familiers avec le concept d'*arbre des suffixes*. Il en définit la structure et détaille l'algorithme de construction présenté par McCreight en 1976. Nous commençons par la description de la structure de *trie des suffixes*, plus simple que celle d'arbre des suffixes mais également plus coûteuse en espace mémoire. Sa présentation nous permet de définir l'arbre des suffixes de manière progressive.

L'*arbre des suffixes* est une structure de données consistant en un indexe compact de tous les facteurs d'un texte. Cette structure peut-être utilisée efficacement pour résoudre le problème classique de recherche de motifs dans un texte. Les méthodes habituelles de recherche d'un motif dans un texte effectuent un pré-traitement du motif et sont alors capables de rechercher le motif en $O(n)$ étapes où n est la longueur du texte [KMP77, BM77, CP91, CR94, Ste94]. Le pré-traitement du motif se fait en temps $O(m)$, m étant la longueur du motif. Si plusieurs motifs doivent être recherchés dans le même texte, chacun des motifs devra d'abord être pré-traité et l'ensemble de toutes les recherches prendra ensuite un temps $O(k.n)$ où k est le nombre de motifs.

Grâce à l'arbre des suffixes du texte, il est possible de rechercher la présence de n'importe quel motif dans le texte en un temps proportionnel à la longueur du motif. La recherche de plusieurs motifs peut donc se faire en un temps proportionnel à la somme des longueurs des motifs. De plus, l'arbre des suffixes permet de déterminer rapidement les facteurs répétés du texte.

Une idée similaire à celle de l'arbre des suffixes a d'abord été présentée implicitement par Morrison lorsqu'il a défini le *"Patricia Tree"* [Mor68],[Knu73, pp. 490-493] (Practical Algorithm To Retrieve Information Coded In Alphanumeric) ; mais la description explicite de l'arbre des suffixes tel qu'il est utilisé à l'heure actuelle a été donnée par Weiner en 1973 [Wei73]. Dans cet article, un algorithme de construction de l'arbre des suffixes en $O(n)$ étapes est proposé. Cet algorithme procède par insertions, dans l'arbre des suffixes, de tous les suffixes du texte, en allant du plus court au plus long. La méthode de McCreight [McC76], que nous décrivons dans ce papier, possède une structure analogue à celui de Weiner mais il insère les suffixes en commençant par le plus long et en terminant par le plus court. Son temps d'exécution est également en $O(n)$ et son intérêt provient surtout de l'économie en espace réalisée par rapport à celui de Weiner.

Plusieurs efforts ont aussi été faits pour développer des algorithmes de construction *en ligne*, c'est-à-dire construisant l'arbre au fur et à mesure de la lecture, de gauche à droite, des symboles du texte [MR80, Ukk92, Ukk95]. Parmi ceux-ci, celui de Ukkonen est le seul capable de construire l'arbre en temps $O(n)$. Cet algorithme étant relativement complexe et puisque la construction en ligne n'est pas utile dans notre cas, nous ne l'aborderons pas. Nous présentons ici l'algorithme de McCreight qui allie simplicité et efficacité. Les ouvrages [CR94, Ste94] présentent et comparent les principaux algorithmes de construction d'arbres des suffixes.

1 Définitions formelles et notations

Soit $\mathcal{A} = \{a_1, a_2, \ldots, a_z\}$ un *alphabet*. C'est un ensemble fini, ses éléments $a_1, a_2, \ldots, a_z$ sont appelés des *lettres*, *symboles* ou *caractères*. Un *mot* x sur l'alphabet $\mathcal{A}$ est une suite finie de lettres de $\mathcal{A}$: $x = x_1 x_2 \ldots x_n$ avec $x_i \in \mathcal{A}, \forall i : 1 \leq i \leq n$. On dit également que x est une *séquence*. La *longueur* de x est notée $|x|$, c'est le nombre de ses lettres : $|x| = n$. Par convention, le *mot vide* est noté $\epsilon : |\epsilon| = 0$.

L'*ensemble de tous les mots* que l'on peut former à partir de l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^*$. C'est un ensemble infini. On définit également l'*ensemble de tous les mots non vides* sur $\mathcal{A} : \mathcal{A}^+ = \mathcal{A}^* \backslash \{\epsilon\}$. Soit $X \subseteq \mathcal{A}^*$ un ensemble de mots, sa cardinalité est notée $\#X$.

Soit $x = x_1 x_2 \ldots x_n$ et $y = y_1 y_2 \ldots y_m$ deux mots de $\mathcal{A}^*$. Leur *concaténation*, notée $x.y$ est le mot de $\mathcal{A}^*$ obtenu en faisant suivre les lettres de x par les lettres de y : $x.y = x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$, sa longueur est $|xy| = |x| + |y| = n + m$. La concaténation de k copies du même mot x est notée x^k.

Soit $x, y \in \mathcal{A}^*$. Le mot y est un *facteur* de x si il existe deux mots $u, v \in \mathcal{A}^*$ tel que $x = uyv$. De plus, si $u = \epsilon$, on dira que y est *préfixe* de x. De la même manière, y est *suffixe* de x si $v = \epsilon$. En d'autre termes, y est facteur de x s'il est égal à une suite contiguë de lettres de x. On note $y = x_{i..j}$ lorsque $y = x_i x_{i+1} \ldots x_j$. Si la suite de lettre contiguës est extraite du début de x, alors y est préfixe de x, on note $y = x_{..i}$ si $y = x_1 x_2 \ldots x_i$. Lorsque y est extrait de la fin de x, y est suffixe de x, on note $y = x_{i..}$ lorsque $y = x_i x_{i+1} \ldots x_n$. Le préfixe $y = x_{..i}$ de x est *préfixe propre* s'il n'est pas égal à $x : i < |x|$. De la même manière, un *suffixe propre* de x est un suffixe de x, différent de x.

On dit que le mot y possède une *occurrence* dans x à la position i si $y = x_{i..j}$. Ce concept permet de définir une *répétition* : le facteur y de x est *répété* s'il possède plusieurs plusieurs occurrences dans x. En d'autres mots, y est une répétition si $y = x_{i_1..(i_1+l-1)} = x_{i_2..(i_2+l-1)} = \ldots$, avec $1 \leq i_1 < i_2 \ldots \leq n$ et $|y| = l$.

Les concepts introduits dans cette section sont illustrés par l'exemple 1.

Exemple 1 *Considérons l'alphabet* $\mathcal{A} = \{a, b, c, d, e\}$ *et la séquence* $x \in \mathcal{A}^*$ *:*

$$x = abceddaabaadeaaaccdabdeabaadeaadcee, |x| = 35$$

Le mot $y = abaade$ *est un facteur répété de* x*, il possède deux occurrences aux positions 8 et 24 (ce sont les zones soulignées). Soit* $u = abced$*. La concaténation de* u *et* y *est le mot* $u.y = abcedabaade$*. Le mot* u *est préfixe de* x *:* $u = x_{..5}$*. Le mot* $v = deaadcee$ *est suffixe de* x *:* $v = x_{28..}$.

2 Préliminaires sur les arbres

Soient p et f deux *nœuds* d'un arbre. Si un *arc* les relie, il est noté $arc(p, f)$. Cet arc est orienté de p vers f ; on dira que p est le *père* et f le *fils* : $p = pere(f)$. Pour le nœud p, $arc(p, f)$ est un arc *sortant* mais pour f, c'est un arc *entrant*. La *racine* est le seul nœud n'ayant pas de père. Tous les autres nœuds ont un et un seul père mais un nœud peut avoir plusieurs fils. On note $fils(p)$ l'ensemble des nœuds fils de p : $f \in fils(p) \iff p = pere(f)$.

Par exemple, la figure 1 représente un arbre à 15 nœuds. Le nœud g_2 est le père des nœuds g_4, g_5 et g_6.

Le nœud g est *ancêtre* du nœud h si $h \in fils(g)$ ou si $h \in fils(f)$ avec g ancêtre de f (définition récursive). De cette manière, la racine est ancêtre de tous les autres nœuds de l'arbre. Pour notre exemple, g_2 est ancêtre de $g_4, g_5, g_6, g_8, g_9, g_{10}, g_{13}$ et g_{14}. Si g est ancêtre de h, alors on dit

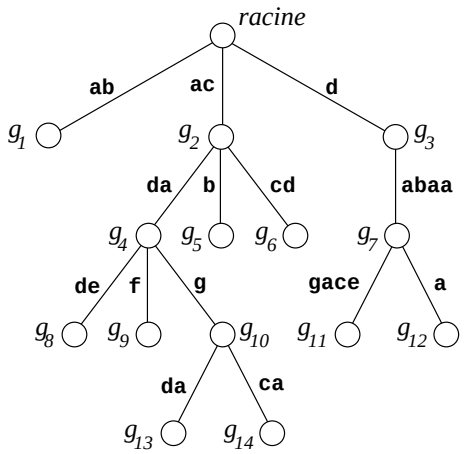

FIG. 1 – Exemple d'arbre

également que h est *descendant* de g. Tout nœud est descendant de la racine excepté la racine elle-même.

Le *degré* d'un nœud p est le nombre de ses fils, on le note $degre(p)$. Un nœud f est une *feuille* s'il ne possède aucun fils : $degre(f) = 0$. Les nœuds de degré supérieur ou égal à 1 sont appelés des *nœuds internes*. Sur l'exemple, $g_1, g_5, g_6, \ldots$ sont des feuilles et $degre(g_4) = 3$.

Un *chemin* est une suite d'arcs consécutifs qui relient, pour chacun, un père avec un de ses fils. Ainsi, la suite $(a_1, a_2, \ldots a_n)$ d'arcs d'un arbre forme un chemin si $a_1 = arc(p_1, p_2), a_2 = arc(p_2, p_3), \ldots a_{n-1} = arc(p_{n-1}, p_n)$ et $a_n = arc(p_n, p_{n+1})$ avec $p_1, p_2, \ldots p_{n+1}$ des nœuds de l'arbre tels que $p_1 = pere(p_2), p_2 = pere(p_3), \ldots p_n = pere(p_{n+1})$. Par définition d'un arbre (il ne contient pas de "boucles"), s'il existe un chemin entre un nœud g et un nœud h, alors ce chemin est unique. On le notera $chemin(g, h)$. Le nœud g sera donc ancêtre de h si et seulement si $chemin(g, h)$ existe. Sur notre exemple, $chemin(g_2, g_{14})$ existe mais ce n'est pas le cas pour $chemin(g_2, g_7)$.

Chacun des arcs de l'arbre peut être étiqueté par un mot. L'étiquette de l'arc $arc(p, f)$ est notée $etiq(arc(p, f))$. Dans notre exemple, les étiquettes des arcs sont indiquées juste à côté des arcs. Par extension, l'étiquette d'un chemin est la concaténation des étiquettes de tous les arcs qui le composent : $etiq(chemin(g, h)) = etiq(arc(g, g')).etiq(arc(g', g'')) \ldots etiq(arc(g^{(i)}, h))$. Le chemin $chemin(g_2, g_{14})$ de notre exemple est étiqueté dagca.

Puisqu'un nœud ne possède qu'un seul père, que la racine ne possède pas de père et que chaque arc représente un lien *père→fils*, le nombre d'arcs d'un arbre est égal au nombre de nœuds moins 1.

3 Le trie des suffixes

Étant donné un ensemble E de mots sur un alphabet $\mathcal{A}$, son *trie* (pour "information re*trie*val", sans traduction en français) [Knu73] est l'arbre possédant les six propriétés suivantes.

1. Chaque arc est étiqueté par un symbole de $\mathcal{A}$.
2. Les étiquettes des arcs menant d'un nœud père vers ses nœuds fils sont toutes différentes : pour tout nœud p, si $f_1, f_2 \in fils(p)$ et $etiq(arc(p, f_1)) = etiq(arc(p, f_2))$, alors $f_1 = f_2$.
3. Il existe deux types de nœuds : les nœuds *terminaux* et les nœuds *non terminaux*. Les feuilles de l'arbre sont toujours des nœuds terminaux.

4. L'étiquette de tout chemin menant de la racine à un nœud terminal est un mot de E : pour tout g, nœud terminal, $etiq(chemin(racine, g)) \in E$.
5. Tout mot de E est l'étiquette d'un chemin allant de la racine à un nœud terminal de l'arbre : pour tout mot w de E, il existe un nœud terminal g de l'arbre tel que $etiq(chemin(racine, g)) = w$.
6. L'étiquette de tout chemin menant de la racine à un nœud non terminal est un préfixe d'un mot de E : pour tout g, nœud non terminal, il existe un mot $w \in E$ tel que $etiq(chemin(racine, g))$ est préfixe de w.

La figure 2 représente le trie de $E = \{A, CA, GC, GG, GGA, GGGCA, GTG, GTT, TC, TCA, TCT\}$ sur l'alphabet $\mathcal{N} = \{A, C, G, T\}$. Les nœuds terminaux sont représentés par des cercles grisés.

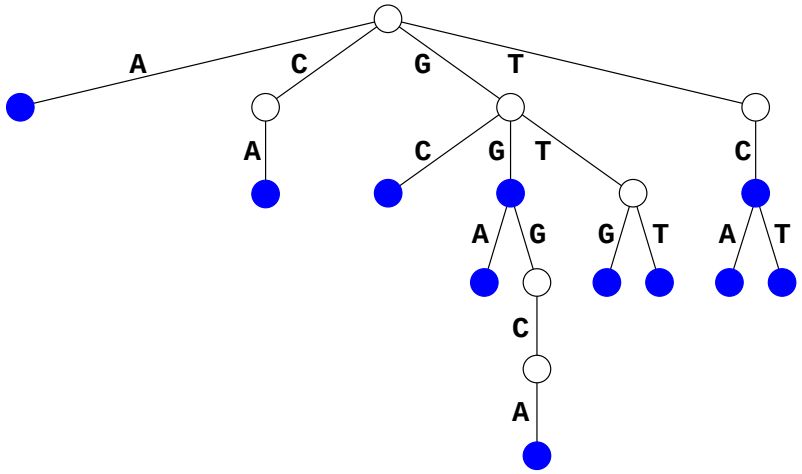

FIG. 2 – Trie de l'ensemble E

Étant donné le mot $y = y_1 y_2 \dots y_m$, on vérifie son appartenance à E en vérifiant s'il est l'étiquette d'un chemin de la racine à un nœud terminal. Pour cela, il suffit, à partir de la racine, de "se déplacer" dans l'arbre en suivant les arcs imposés par les symboles de y, lus de gauche à droite. Si le chemin n'existe pas parce qu'un nœud intermédiaire ne possède pas le fils adéquat, alors y n'appartient pas à E. C'est le cas, dans l'exemple, avec $y = GTA$. Si le chemin conduit à un nœud non terminal, alors y n'appartient pas à E mais est préfixe d'un mot de E. Par exemple, $y = GGG$ est préfixe du mot $GGGCA$ de E. Si le chemin atteint un nœud terminal, alors y appartient à E. La vérification d'appartenance se fait donc en temps $O(m)$.

On remarque facilement que si tous les mots de E se terminent par le même symbole n'apparaissant nulle part ailleurs dans les mots de E, alors tous les nœuds terminaux sont des feuilles. En effet, aucun mot de E n'est le préfixe d'un autre mot de E.

Soit $x = x_1 x_2 \dots x_n \in \mathcal{A}^+$ un mot. Supposons que son dernier symbole soit différent de tous les autres. Si ce n'est pas le cas, il suffit d'étendre l'alphabet à $\mathcal{A} \cup \{\$\}$, avec $\$ \notin \mathcal{A}$, et de considérer le mot x auquel on concatène le symbole $\$$. Le *trie des suffixes* de x, noté $STrie(x)$, est le trie de l'ensemble S_x constitué de tous les suffixes de x.

Si $x = ACTACT\$$, alors $S_x = \{ACTACT\$, CTACT\$, TACT\$, ACT\$, CT\$, T\$, \$\}$ et $STrie(x)$ est l'arbre représenté à la figure 3.

Puisque le dernier symbole de x est différent de tous les autres, aucun suffixe n'est le préfixe d'un autre suffixe et donc les nœuds terminaux de $STrie(x)$ sont tous situés sur les feuilles. Dès lors, plutôt

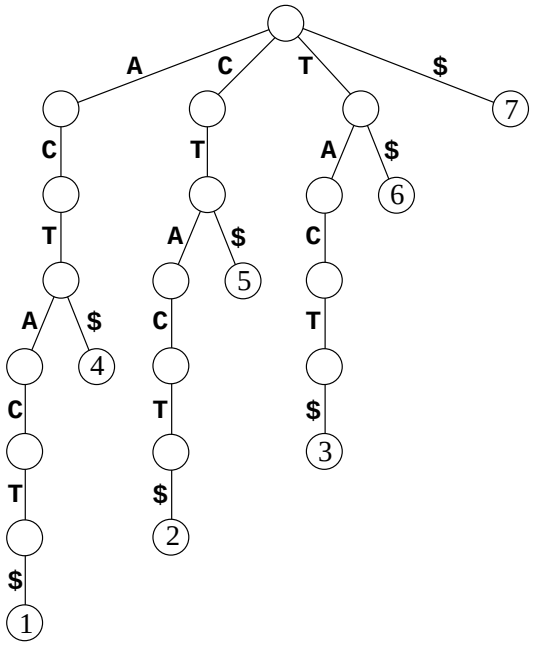

FIG. 3 – STrie(ACTACT$)

que de représenter les nœuds terminaux par des cercles grisés, nous plaçons, dans chaque feuille, la position de début du suffixe concerné.

Étant donné un mot $y = y_1 y_2 \dots y_m$, on peut tester, en maximum m étapes, s'il est facteur de x. En effet, s'il correspond à un chemin partant de la racine et se terminant sur une feuille, c'est un suffixe de x. Si le chemin se termine sur un nœud interne, alors y est préfixe d'un suffixe de x et donc facteur de x. Si le chemin n'existe pas, alors y n'est pas facteur de x.

Donc, il y a non seulement correspondance bi-univoque entre les feuilles et les suffixes de x mais, de plus, les nœuds internes correspondent aux facteurs de x. Pour tout nœud g, on peut donc définir $\mathtt{fact}(g)$ comme l'étiquette du chemin menant de la racine à g. Par définition, $\mathtt{fact}(racine) = \epsilon$. Symétriquement, étant donné un mot w, son *locus* est le nœud g du trie des suffixes pour lequel $\mathtt{etiq}(\mathtt{chemin}(racine, g)) = w : \mathtt{locus}(\mathtt{fact}(g)) = g$. Il est important de remarquer que le locus n'existe dans le trie que pour les facteurs du mot.

La proposition 1 montre que l'information d'un nœud de degré 1 peut être immédiatement déduite de l'information de son unique fils. Par contre, la proposition 2 montre qu'un nœud interne de degré supérieur ou égal à 2, ou une feuille, apporte de l'information supplémentaire. De ce fait, ces derniers nœuds sont communément appelés *nœuds importants* [CR94, Riv96].

Proposition 1 *Soit x un mot. Soient g et f deux nœuds de $\mathtt{STrie}(x)$ tels que $\mathtt{degre}(g) = 1$ et $\mathtt{pere}(f) = g$. Supposons que $\mathtt{fact}(f)$ soit connu, alors grâce à $\mathtt{etiq}(\mathtt{arc}(g, f))$ on déduit immédiatement $\mathtt{fact}(g)$.*

Preuve. Par construction de $\mathtt{STrie}(x)$, on a $\mathtt{fact}(g).\mathtt{etiq}(\mathtt{arc}(g, f)) = \mathtt{fact}(f)$. □

Proposition 2 *Soit $x = x_1 x_2 \dots x_n$ un mot, avec $x_n \neq x_i, \forall i : 1 \leq i < n$.*

1. *Soit h une feuille de $\mathtt{STrie}(x)$. Alors $\mathtt{fact}(h)$ est un suffixe de x.*

2. *Soit* g *un nœud de* $\mathsf{STrie}(x)$ *avec* $\mathtt{degre}(g) \geq 2$ *et* $l = |\mathtt{fact}(g)|$. *Soient* $f_1, f_2 \in \mathtt{fils}(g)$ *avec* $\mathtt{etiq}(\mathtt{arc}(g, f_1)) \neq \mathtt{etiq}(\mathtt{arc}(g, f_2))$. *Il existe deux suffixes* $x_{i..}$ *et* $x_{j..}$ *de* x *tels que :*
   - $\mathtt{fact}(g)$ *est le préfixe de longueur* l *à la fois de* $x_{i..}$ *et de* $x_{j..}$.
   - $\mathtt{fact}(f_1)$ *est le préfixe de longueur* $l+1$ *de* $x_{i..}$ *mais n'est pas préfixe de* $x_{j..}$.
   - $\mathtt{fact}(f_2)$ *est le préfixe de longueur* $l+1$ *de* $x_{j..}$ *mais n'est pas préfixe de* $x_{i..}$.

Preuve.

1. $\mathtt{fact}(h)$ est un suffixe de x par définition de $\mathsf{STrie}(x)$.
2. Les mots $\mathtt{fact}(g)$, $\mathtt{fact}(f_1)$ et $\mathtt{fact}(f_2)$ sont tous trois préfixes de suffixes de x par construction de $\mathsf{STrie}(x)$. Donc il existe $i \leq j \leq n$ tels que $\mathtt{fact}(f_1) = x_{i..i+l}$ et $\mathtt{fact}(f_2) = x_{j..j+l}$. Puisque $\mathtt{fact}(f_1) = \mathtt{fact}(g).\mathtt{etiq}(\mathtt{arc}(g, f_1))$, $\mathtt{fact}(g)$ est le préfixe de longueur l de $x_{i..}$. De la même façon, $\mathtt{fact}(g)$ est le préfixe de longueur l de $x_{j..}$.
   Puisque $l = |\mathtt{fact}(g)|$, on a $x_{i..i+l-1} = x_{j..j+l-1}$. Puisque $x_{i+l} = \mathtt{etiq}(\mathtt{arc}(g, f_1)) \neq \mathtt{etiq}(\mathtt{arc}(g, f_2)) = x_{j+l}$, $x_{i..i+l}$ n'est pas préfixe de $x_{j..}$ et inversement. □

Chaque nœud interne, de degré supérieur ou égal à 2 correspond donc à un facteur répété *maximal à droite*. On entend par là que chacune des occurrences de la répétition possède un contexte droit différent : $x_{i..i+l-1} = x_{j..j+l-1}$ mais $x_{i+l} \neq x_{j+l}$. La répétition ne peut donc pas être étendue à droite pour chacune des deux occurrences.

Le trie des suffixes est un outil très puissant mais malheureusement, coûteux en espace mémoire. En effet, dans le pire des cas, par exemple lorsque tous les symboles de x sont différents, il n'existe aucun préfixe commun à deux suffixes quelconques de x. Dès lors, le nombre d'arcs du trie sera égal à la somme des longueurs de tous les suffixes de x, c'est-à-dire $n + (n-1) + (n-2) \ldots + 1 = \frac{n^2+n}{2}$ (voir figure 4).

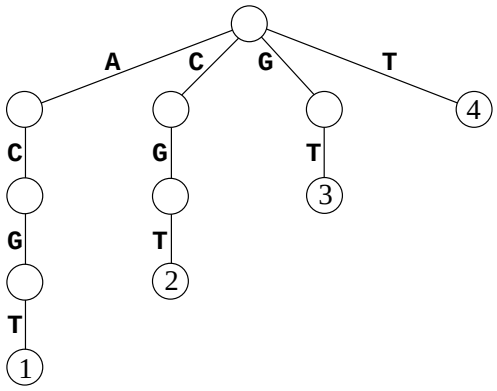

FIG. 4 – $\mathsf{STrie}(\mathsf{ACGT})$

4 Du trie des suffixes à l'arbre des suffixes

La séparation des nœuds du trie des suffixes en nœuds *importants* et autres nœuds permet d'en réduire le nombre de nœuds.

Soit g un nœud de degré 1 de $\mathsf{STrie}(x)$. Soit $p = \mathtt{pere}(g)$. Soit h le plus proche nœud important descendant de g. Ce nœud existe et est unique, dans le pire des cas, c'est une feuille. Le chemin $\mathtt{chemin}(p, h)$ ne contient que des nœuds de degré 1 : $\mathtt{chemin}(p, h) = (\mathtt{arc}(p, g), \mathtt{arc}(g, f_1), \mathtt{arc}(f_1, f_2), \ldots, \mathtt{arc}(f_i, h))$ et $\mathtt{degre}(g) = \mathtt{degre}(f_1) = \ldots = \mathtt{degre}(f_i) = 1$ (voir figure 5).

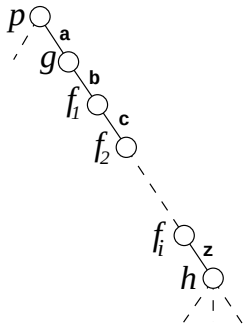

FIG. 5 – Chemin composé de nœuds de degré 1

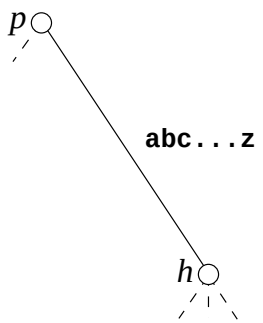

FIG. 6 – Chemin de la figure 5 compacté

Grâce à la proposition 1, l'information de g peut être déduite de l'information de f_1 qui peut elle-même être déduite de l'information de $f_2 \ldots$ l'information de f_i qui peut être déduite de l'information de h. Dès lors, on compacte $chemin(p,h)$ en un seul arc en supprimant tous les nœuds intermédiaires de degré 1. L'étiquette du nouvel arc est la concaténation des étiquettes des arcs supprimés : $etiq(arc(p,h)) = etiq(arc(p,g)).etiq(arc(g,f_1)) \ldots etiq(arc(f_i,h))$ (voir figure 6).

L'itération de ce compactage sur tous les chemins entièrement composés de nœuds de degré 1 produit l'*arbre des suffixes*, noté $ST(x)$. Dans $ST(x)$, tous les nœuds internes sont de degré supérieur ou égal à 2 (excepté éventuellement la racine d'un arbre des suffixes trivial). Chaque arc de $ST(x)$ est étiqueté par un facteur de x. Étant donné un nœud p de $ST(x)$, $fact(p)$ est égal à la concaténation des étiquettes des arcs constituant le chemin $chemin(racine, p)$.

Par construction, pour un nœud interne p de $ST(x)$, les étiquettes des arcs menant à chacun de ses fils commencent toutes par un symbole différent.

Bien entendu, la diminution du nombre de nœuds a provoqué un accroissement de la quantité d'information à stocker pour chaque arc (l'étiquette peut contenir plusieurs lettres). Si on suppose la connaissance de x, l'étiquette de chaque arc $arc(p,h)$ peut être remplacée par le couple (i,j) tel que $etiq(arc(p,h)) = x_{i..j}$.

Remarque 1 *Puisqu'un nœud de degré supérieur ou égal à 2 correspond à un facteur répété, le couple* (i,j) *est choisi arbitrairement parmi les positions de début et de fin des occurrences de la répétition.*

La figure 7 présente l'arbre des suffixes correspondant au trie des suffixes de la figure 3. Les étiquettes des arcs figurent sur le graphique tout comme les couples de positions (i,j) qui délimitent les facteurs.

La proposition suivante établit une borne au nombre de nœuds de l'arbre des suffixes.

Proposition 3 *Soit* $x = x_1x_2 \ldots x_n$. *le nombre maximal de nœuds de* $ST(x)$ *est* $2n-1$.

Preuve. Puisqu'il y a n suffixes, $ST(x)$ contient exactement n feuilles. Par compactage de $STrie(x)$, tous les nœuds internes de $ST(x)$ ont un degré supérieur ou égal à 2. Il y a donc au plus $\lfloor \frac{n}{2} \rfloor$ pères de feuilles. Il y a au plus $\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor$ "grand-pères" de feuilles... L'arbre $ST(x)$ contient donc au maximum $\lfloor \frac{n}{2} \rfloor + \left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor}{2} \right\rfloor \ldots + 1 \leq n-1$ nœuds internes. La comptabilisation des feuilles donne

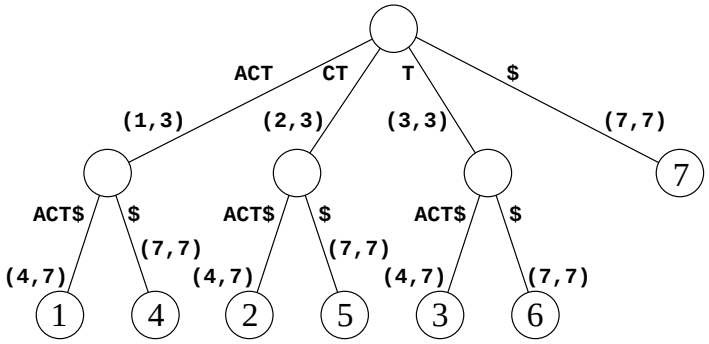

FIG. 7 – $ST(\mathsf{ACTACT\$})$

$2n-1$ nœuds au maximum. □

Soit $y = x_{i..j}$ un facteur de x. Si, dans $STrie(x)$, $locus(y)$ était un nœud de degré 1, alors $locus(y)$ n'existe pas dans $ST(x)$. On définit alors le *locus contracté* du facteur y dans $ST(x)$ comme étant le locus du plus long préfixe de y représenté dans $ST(x)$. Sur l'exemple de la figure 7, le locus contracté de $y = \mathsf{ACTA}$ est $locus(\mathsf{ACT})$. Le *locus étendu* de y est le locus du plus court facteur de x, représenté dans $ST(x)$, dont y est un préfixe. Sur l'exemple de la figure 7, le locus étendu de $y = \mathsf{CTA}$ est $locus(\mathsf{CTACT\$})$, c'est-à-dire la feuille 2.

Tout comme pour le trie des suffixes, vérifier si $y = y_1y_2\dots y_m$ est un facteur de x, se fait en temps $O(m)$. Il suffit, à partir de la racine, de descendre parmi les fils en suivant les arcs imposés par les symboles de y, lus de gauche à droite. Chaque descente d'un nœud père p vers son fils f impose la lecture, dans y de $|etiq(arc(p,f))|$ symboles. À chaque nœud, six cas peuvent se produire :

1. Il y a concordance entre le facteur $etiq(arc(p,f))$ et les $|etiq(arc(p,f))|$ symboles courants de y. Dans ce cas, **la descente doit continuer** vers le fils f. C'est le cas du parcours de ACT pour $y = \mathsf{ACTA}$ dans l'exemple de la figure 7.
2. Le nœud courant est une feuille et les symboles de y ont tous été parcourus. On conclut que y **est un suffixe de** x. C'est le cas de $y = \mathsf{ACT\$}$ après la descente de deux arcs pour notre exemple.
3. Le nœud courant n'est pas une feuille mais tous les symboles de y ont été parcourus. Alors y **est un facteur répété maximal à droite de** x. C'est le cas de $y = \mathsf{ACT}$ après la descente d'un arc pour notre exemple.
4. Tous les symboles de y ont été parcourus mais seuls les premiers symboles de l'étiquette de l'arc courant ont été lus. Dans ce cas, y **est un facteur de** x, le dernier nœud parcouru dans $ST(x)$ est le locus contracté de y. C'est le cas de $y = \mathsf{ACTAC}$ après descente d'un arc et parcours des deux symboles AC qui terminent y.
5. La comparaison des symboles courants de y avec ceux du facteur $etiq(arc(p,f))$ provoque une discordance. Le mot y **n'est alors pas un facteur de** x. C'est le cas de $y = \mathsf{ACA}$ lors de la descente dans le premier arc.
6. Le nœud courant ne possède pas le fils correspondant au caractère courant de y. Le mot y **n'est alors pas un facteur de** x. C'est le cas de $y = \mathsf{ACTCA}$ après la descente dans le premier arc : $locus(\mathsf{ACT})$ ne possède pas de fils pour le symbole C.

L'information de l'arbre des suffixes permet donc de tester si un mot est un facteur de x en un temps équivalent au test utilisant le trie des suffixes.

5 Structure de données pour le stockage de l'arbre des suffixes

Puisqu'on travaille avec des arbres, chaque nœud (excepté la racine) est le fils d'un et d'un seul père. Dès lors, l'étiquette d'un arc (en fait les deux indices, dans χ, qui délimitent l'étiquette de l'arc) peut être stockée dans le nœud fils de chaque arc *père*→*fils*.

Pour le stockage des arcs eux-mêmes, deux cas sont à distinguer :

1er cas : L'alphabet contient un nombre indéterminé ou un grand nombre de lettres. Dès lors, le nombre de fils peut varier fortement d'un nœud à l'autre.

Une structure possible est celle dans laquelle chaque nœud possède un pointeur vers son premier fils et un pointeur vers son frère droit. La figure 8 montre cette structure de données pour l'arbre des suffixes de la figure 7 dans le cas où l'alphabet n'est pas connu ou comporte trop de lettres. Chaque pointeur *nul* est représenté par λ. L'accès à un fils se fait en temps $O(\#\mathcal{A})$ puisqu'il faut

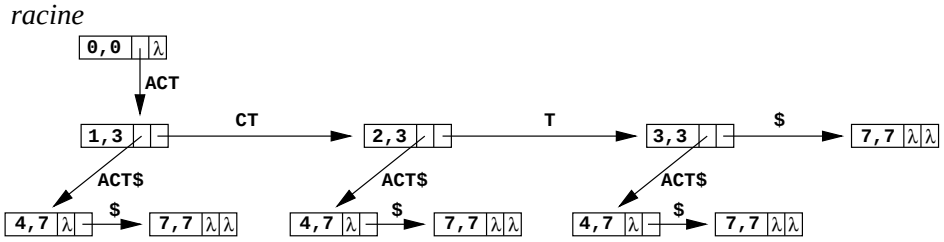

FIG. 8 – Structure de données pour un alphabet inconnu

rechercher le fils adéquat dans la liste des fils. Dans ce cas, le temps maximal de recherche d'un facteur dans l'arbre des suffixes est à multiplier par $O(\#\mathcal{A})$.

Une autre possibilité est de placer tous les fils d'un même nœud dans une structure de recherche permettant l'accès à un fils en temps $O(\log \#\mathcal{A})$.

2ème cas : L'alphabet est déterminé, il contient un petit nombre de lettres. C'est le cas par exemple des séquences d'ADN : $\mathcal{N}_\$ = \mathcal{N} \cup \{\$\} = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \$\}$. Dans ce cas, chaque nœud peut contenir un pointeur pour chacun de ses fils potentiels. Puisque l'étiquette de chaque arc commence par une lettre différente, il faut prévoir un pointeur par lettre de l'alphabet. On suppose alors que l'accès à un fils se fait en temps constant.

La figure 9 montre la structure de données associée à l'arbre des suffixes de la figure 7 dans le cas où l'alphabet est déterminé et contient peu de lettres (en l'occurrence, $\mathcal{N}_\$ = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \$\}$).

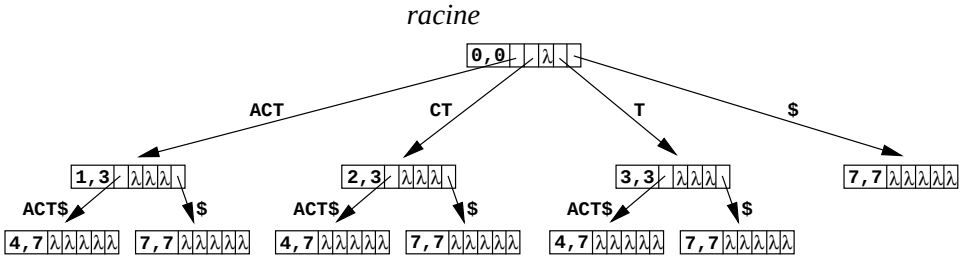

FIG. 9 – Structure de données pour un alphabet petit et déterminé

6 Utilisation pour localiser les facteurs répétés dans une séquence

Puisque chaque nœud interne de l'arbre des suffixes $ST(x)$ est de degré supérieur ou égal à deux, c'est un facteur répété maximal à droite : chacune des occurrences de la répétition possède un contexte droit différent. La répétition ne peut pas être étendue à droite.

Grâce à l'arbre des suffixes, il est aisé de localiser toutes les occurrences d'un facteur maximal à droite.

Soit f un facteur répété maximal à droite, il possède un locus dans l'arbre : $g = locus(f)$. Soient $g_1, g_2, \ldots g_k$ toutes les feuilles du sous-arbre dont g est la racine :

- Ce sont des feuilles : $\forall g_i : 1 \leq i \leq k : degre(g_i) = 0$.
- g est leur ancêtre commun : $\forall g_i : 1 \leq i \leq k :$ g est ancêtre de g_i.
- Toutes les feuilles sont considérées : il n'existe pas de feuille h de l'arbre telle que g est ancêtre de h et $h \notin \{g_1, g_2, \ldots g_k\}$.

Chaque nœud g_i est le locus d'un suffixe de x. Par construction de l'arbre des suffixes, $fact(g)$ est préfixe de chacun des suffixes $fact(g_1), fact(g_2), \ldots fact(g_k)$ et par conséquent, les positions d'occurrence du facteur répété $fact(g)$ sont les positions de début des suffixes $fact(g_i)$. Étant donné un nœud interne, les positions de ses occurrences sont trouvées par recherche récursive des positions des occurrences de toutes les feuilles dont il est ancêtre.

Supposons qu'un facteur f répété ne soit pas maximal à droite, alors les positions de chacune de ses occurrences sont égales aux positions des occurrences du facteur de son locus étendu.

Les facteurs répétés maximaux sont importants dans le contexte de la compression. On code en effet la deuxième (ou une des suivantes) occurrence d'un facteur répété par un pointeur qui identifie la première occurrence (pour plus de précisions, voir le chapitre suivant). Plus long est le facteur répété, meilleur est le gain produit par le codage du pointeur. Un facteur répété maximal à droite est donc potentiellement plus intéressant qu'un des facteurs répétés qu'il contient comme facteur.

Il est possible de déterminer la liste des positions de toutes les occurrences des facteurs répétés maximaux à droite par simple parcours récursif de l'arbre. Chaque nœud est visité une seule fois, le processus donc linéaire en temps.

7 Construction de l'arbre des suffixes

Nous présentons ici l'algorithme de McCreight [McC76] qui construit $ST(x_1x_2 \ldots x_n)$ en temps $O(n)$. Avant de détailler cet algorithme qui utilise astucieusement une propriété des suffixes, nous présentons un algorithme intuitif appelé *algorithme en force brute* (pour "brute force" en anglais). Cet algorithme construit $ST(x_1x_2 \ldots x_n)$ en temps $O(n^2)$ mais il permet de présenter l'amélioration de McCreight et de définir certains concepts utiles.

7.1 Algorithme en force brute

Étant donné le mot $x = x_1x_2 \ldots x_n$, avec $\forall i : 1 \leq i < n : x_i \neq x_n$, une idée simple pour construire $ST(x)$ est de partir d'un arbre trivial ne contenant qu'un nœud et d'y insérer successivement les chemins correspondant à tous les suffixes $x_{i..}$ de x.

Considérons la fonction BRUTEFORCEST présentée à la figure 10. Les suffixes de x sont insérés dans l'arbre des suffixes du plus long au plus court. L'ordre dans lequel ils sont insérés n'a pas vraiment d'importance à ce stade-ci mais l'algorithme de McCreight qui sera présenté au point 7.2 exploite spécifiquement cet ordre. Pour cette raison, nous avons choisi ici d'insérer les suffixes en suivant le même ordre.

```
BRUTEFORCEST(x, n)
1  ST_0 ← INITARBRE()
2  Pour i ← 1 Jusque n
3  Faire ST_i ← INSÈRESUF(ST_{i-1}, x_{i..})
4  Retourner ST_n
```

FIG. 10 – Algorithme en force brute

Dans cet algorithme, ST_i désigne l'*arbre partiel des suffixes* de x après la $i^{\text{ème}}$ étape de la construction de $ST(x)$. L'arbre ST_n est l'arbre final $ST(x)$. La fonction INITARBRE crée l'arbre initial ST_0 ne contenant qu'un seul nœud : la racine du futur arbre des suffixes.

Nous détaillons maintenant la fonction INSÈRESUF qui crée l'arbre ST_i par insertion du suffixe $x_{i..}$ dans ST_{i-1}.

Soit $head_i$ le plus long préfixe de $x_{i..}$ qui soit également préfixe d'un suffixe $x_{j..}$ avec $j < i$. Puisque $j < i$, $fact(x_{j..})$ est une feuille de ST_{i-1} et $head_i$ est donc le plus long préfixe de $x_{i..}$ qui possède un locus étendu dans ST_{i-1}. Soit $tail_i$ le mot tel que $x_{i..} = head_i.tail_i$. Puisqu'aucun suffixe de x n'est préfixe d'un autre suffixe de x, $tail_i \neq \epsilon$. La première étape de INSÈRESUF est donc la localisation du locus étendu de $head_i$ dans ST_{i-1}. Si ce locus étendu n'est pas le locus de $head_i$, alors ce dernier doit être créé. Le nouveau nœud servant de locus à $head_i$ est inséré au milieu de l'arc joignant son locus contracté à son locus étendu. Les étiquettes des deux arcs ainsi créés sont obtenues par séparation de l'étiquette de l'ancien arc à l'endroit ou les caractères ne correspondent plus avec ceux de $x_{i..}$. Pour terminer, la feuille $locus(x_{i..})$ est insérée comme fils de $locus(head_i)$, l'arc qui les relie est étiqueté par $tail_i$. L'arbre ST_i est alors construit.

Cette méthode est illustrée à la figure 11 qui présente les étapes successives de la construction de $ST(\mathsf{ACAC\$})$.

Pour réaliser son travail, INSÈRESUF dispose des deux fonctions générales COUPEARC et LOCUSPRÉFIXECOMMUN. Supposons qu'un nœud g possède un arc sortant $arc(g, f)$ tel que le mot w soit préfixe propre de $etiq(arc(g, f))$. L'appel COUPEARC(g, w) crée un nouveau nœud h qui coupe l'arc $arc(g, f)$ en deux arcs $arc(g, h)$ et $arc(h, f)$ de telle sorte que $etiq(arc(g, h)) = w$ et l'ancienne étiquette $etiq(arc(g, f))$ soit égale à $w.etiq(arc(h, f))$ (voir figure 12). Si l'accès à un fils spécifique d'un nœud se fait en temps $O(1)$ (voir 5), alors le travail de COUPEARC se réalise également en temps $O(1)$. COUPEARC retourne le nœud créé.

Soit g la racine d'un arbre des suffixes partiel et w un mot qui n'est le préfixe d'aucun suffixe représenté dans l'arbre. Soit $head$ le plus long préfixe de w qui possède un locus étendu dans l'arbre de racine g : $w = head.tail$. L'appel LOCUSPRÉFIXECOMMUN$(g, w, IndiceTail)$ retourne le locus de $head$ dans l'arbre de racine g (voir figure 13). LOCUSPRÉFIXECOMMUN fait éventuellement appel à COUPEARC pour créer le locus s'il n'existait pas. Le but de cette fonction dans l'algorithme glouton est de rechercher et éventuellement de construire le locus de $head_i$ lors de l'exécution de INSÈRESUF.

La fonction retourne également, via son troisième paramètre $IndiceTail$, l'indice, dans w, du début du mot $tail$. L'algorithme de LOCUSPRÉFIXECOMMUN est présenté à la figure 14. La fonction ACCÈDEFILS accède au fils d'un nœud étant donnée la première lettre de l'arc qui y mène. Si ce fils n'existe pas, elle retourne NIL. À partir de la racine g, le chemin étiqueté par le plus long préfixe de w est parcouru caractère par caractère. Si le processus s'arrête sur un nœud spécifique parce qu'il ne possède pas le fils adéquat, le plus long préfixe a été trouvé et son locus existe déjà. Cela signifie que deux suffixes insérés auparavant dans l'arbre possédaient déjà ce mot comme plus long préfixe

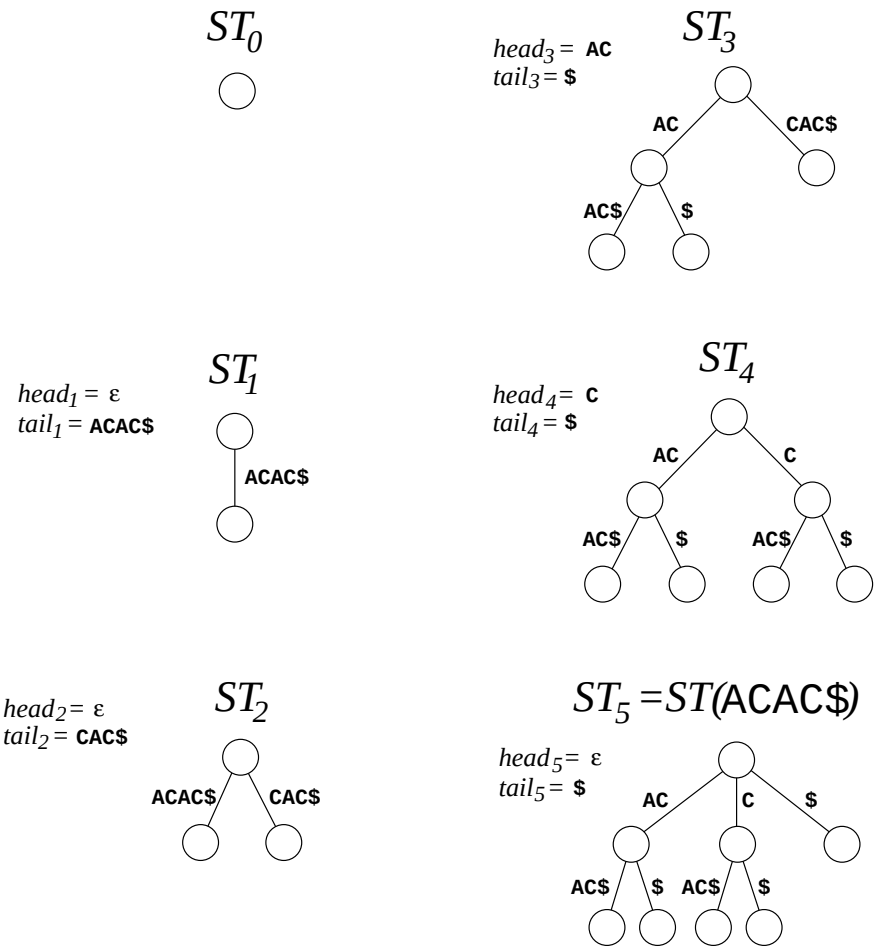

FIG. 11 – Les 5 étapes de la construction de $ST(\texttt{ACAC\$})$

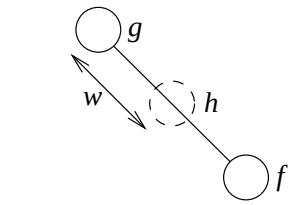

FIG. 12 – COUPEARC(g, w)

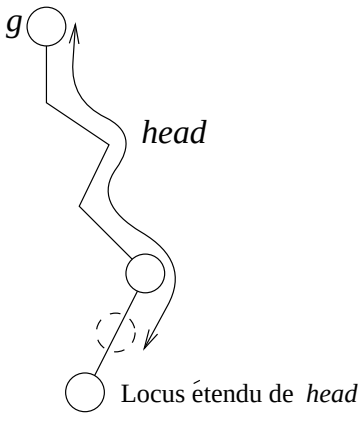

FIG. 13 – LOCUSPRÉFIXECOMMUN$(g, w,$ IndiceTail$)$

```
LocusPréfixeCommun(g, w, IndiceTail)
 1  NœudCourant ← g
 2  i ← 1                                  /* Position courante dans w */
 3  Trouve ← Faux
 4  TantQue Pas(Trouve)
 5  Faire h ← AccèdeFils(NœudCourant, w_i)
 6                                         /* On accède au fils grâce à la première lettre w_i */
 7       Si h = Nil
 8          Alors IndiceTail ← i
 9                Retourner NœudCourant
10          Sinon j ← 1                   /* Position courante dans l'arc */
11                u ← etiq(arc(NœudCourant, h))
12                TantQue (j ≤ |u|) et (u_j = w_i)
13                Faire i ← i + 1         /* Comparaison */
14                      j ← j + 1
15                Si j > |u|              /* Concordance */
16                   Alors NœudCourant ← h
17                   Sinon Trouve ← Vrai
18  IndiceTail ← i
19  Retourner CoupeArc(NœudCourant, w_{i-j+1..i-1})
```

FIG. 14 – Fonction LOCUSPRÉFIXECOMMUN de recherche du locus du plus long préfixe de w présent dans l'arbre de racine g

commun. Si la fin du préfixe est détectée au milieu d'un arc, alors COUPEARC est appelé pour créer le nouveau locus.

Dans le pire des cas, tous les caractères de w, excepté le dernier, seront parcourus. LOCUSPRÉFIXECOMMUN s'exécute donc en temps $O(|w|)$.

Grâce à l'appel $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(racine, x_{i..}, IndiceTail)$, la fonction INSÈRESUF connaît maintenant $locus(head_i)$. Il lui suffit alors de créer la nouvelle feuille correspondant au suffixe $x_{i..}$ et de créer et étiqueter l'arc qui la lie à son père $locus(head_i)$. L'étiquette de l'arc est déterminée par le couple $(IndiceTail, n)$. La création de la feuille et l'arc se fait en temps $O(1)$ par un appel de type CRÉENŒUD$(locus(head_i), IndiceTail, n)$ où CRÉENŒUD(p, i_1, i_2) crée un fils du nœud p, l'étiquette du nouvel arc étant déterminée par le couple d'indices (i_1,i_2) dans x.

La fonction LOCUSPRÉFIXECOMMUN étant appelée pour chaque suffixe $x_{i..}$ de x, l'algorithme glouton fonctionne en temps $O(|x_{1..}| + |x_{2..}| \ldots + 1) = O(n^2)$. Cette borne maximale est atteinte par exemple pour $x =$ AAAAAAAAAA$.

7.2 Algorithme de McCreight

L'algorithme de McCreight [McC76] suit le même schéma que l'algorithme en force brute : les suffixes de x sont insérés dans $ST(x)$ en allant du plus long $(x_{1..})$ au plus court $(x_{n..})$. À chaque étape i, l'algorithme recherche (et éventuellement construit) le nœud $locus(head_i)$ et crée ensuite la feuille correspondant au suffixe $x_{i..}$. Pour gagner du temps lors de la recherche de $head_i$, l'algorithme de McCreight utilise une propriété qui lie le suffixe $x_{i..}$ au suffixe précédent $x_{i-1..}$.

Nous commençons la présentation par deux propriétés générales de l'arbre des suffixes qui seront souvent utilisées par la suite. La première proposition présentée découle immédiatement du schéma d'insertion des suffixes.

7.2.1 Propriétés des nœuds internes

Proposition 4 *Tout nœud interne de* $ST(x)$ *a été créé à une étape* i *parce qu'il était le locus de* $head_i$.

Preuve. Vrai par construction de $ST(x)$. □

La proposition 5 permet de définir, pour chaque nœud interne, son *lien suffixe* qui permettra de court-circuiter la recherche des $head_i$.

Proposition 5 *Si un nœud interne* g, *locus de* $a.\gamma$, *avec* $a \in \mathcal{A}$ *et* $\gamma \in \mathcal{A}^*$ *est un nœud interne de* $ST(x)$ *alors le nœud* h, *locus de* γ, *existe et est également un nœud interne de* $ST(x)$.

Preuve. Soit $l = |a\gamma|$. D'après la proposition 4, il existe i tel que $head_i = a\gamma$. Cela signifie qu'il existe $j < i$ tel que $a\gamma$ est le plus long préfixe commun entre $x_{i..}$ et $x_{j..} : x_{i..i+l-1} = x_{j..j+l-1} = a\gamma$ mais $x_{i+l} \neq x_{j+l}$. Dans ce cas, γ est le plus long préfixe commun entre $x_{i+1..}$ et $x_{j+1..} : x_{i+1..i+l-1} = x_{j+1..j+l-1} = \gamma$ mais $x_{i+l} \neq x_{j+l}$. Puisque $j+1 < i+1$, lors de l'insertion du suffixe $x_{i+1..}$, il y aura séparation entre $chemin(racine, locus(x_{i+1..}))$ et $chemin(racine, locus(x_{j+1..}))$ au nœud $locus(\gamma)$ (voir figure 15). □

Remarque 2 *Dans la preuve de la proposition 5, le mot* γ *n'est pas forcément égal à* $head_{i+1}$. *En effet, il se peut qu'un suffixe* $x_{k+1..}$ *avec* $k+1 < i+1$ *ait un plus long préfixe commun avec* $x_{i+1..}$ *qui soit plus long que celui que partagent* $x_{i+1..}$ *et* $x_{j+1..}$. *Pour cela, il suffit simplement que* $x_{k..} = b.\gamma.\alpha.\beta$ *et* $x_{i..} = a.\gamma.\alpha.\delta$ *avec* $b \in \mathcal{A}$ *et* $a \neq b$. *Dans ce cas,* $x_{k..}$ *et* $x_{i..}$ *n'ont aucun préfixe en commun mais* $x_{k+1..}$ *et* $x_{i+1..}$ *ont* $\gamma\alpha$ *en commun (voir figure 15).*

Dès lors, tout nœud interne $g = locus(a\gamma)$ *possède un nœud équivalent* $h = locus(\gamma)$. *Cependant, les caractéristiques de* g *et de* h *ne sont pas identiques :* h *peut posséder plus de fils ou plus de descendants que* g. *En effet, soit* T_1 *le sous-arbre dont* g *est la racine. Soit* T *le sous-arbre dont* h *est la racine (voir figure 16). Supposons que* f, *locus de* $b\gamma$ *avec* $b \in \mathcal{A}$ *et* $b \neq a$ *existe et soit racine du sous-arbre* T_2, *alors le sous-arbre* T *est en quelque sorte une "fusion" entre* T_1 *et* T_2.

7.2.2 Liens suffixes

Nous pouvons maintenant ajouter, au stockage de chaque nœud interne $g = locus(a\gamma)$, un pointeur vers le nœud $h = locus(\gamma)$.

L'algorithme de McCreight se sert efficacement de ce nouveau pointeur appelé *lien suffixe*. Il doit également, au fur et à mesure de la création des nœuds internes, donner une valeur correcte à ces liens. On note $h = liensuf(g)$ si $g = locus(a\gamma)$ et $h = locus(\gamma)$. On note également $h = suf(a\gamma)$.

Si à la fin de la construction de l'arbre des suffixes, tous les nœuds internes sauf la racine possèdent un lien suffixe valide, ce n'est pas le cas lors des étapes intermédiaires. Ainsi, la proposition 6 montre que $liensuf(locus(head_{i-1}))$ ne peut recevoir une valeur correcte qu'à la fin de l'étape i.

Proposition 6 *À la fin de l'étape* i, *après insertion de* $x_{i..}$ *dans l'arbre des suffixes intermédiaire, si* $head_{i-1} \neq \epsilon$, *alors la valeur à donner à* $liensuf(locus(head_{i-1}))$ *est connue.*

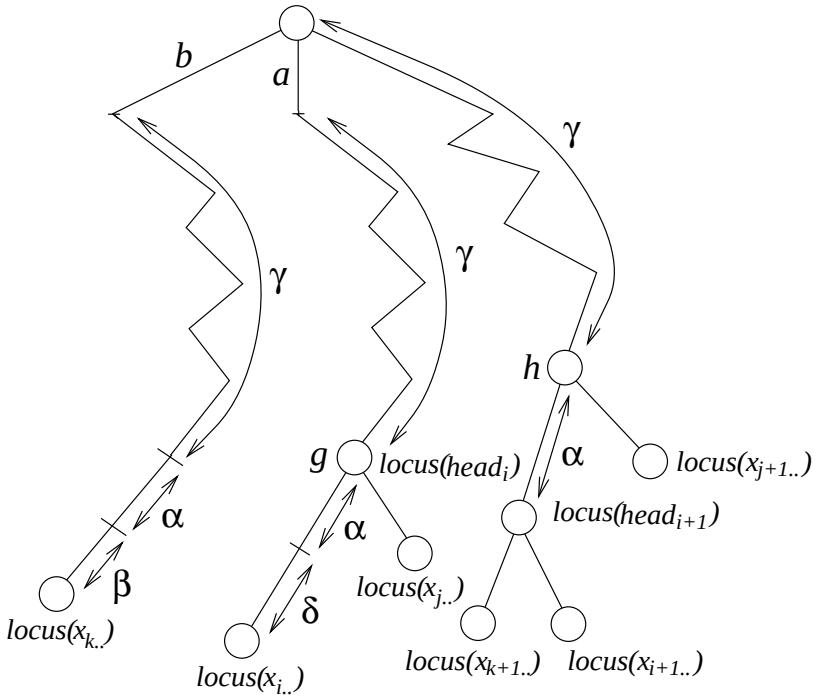

FIG. 15 – Insertion de $x_{i+1..}$

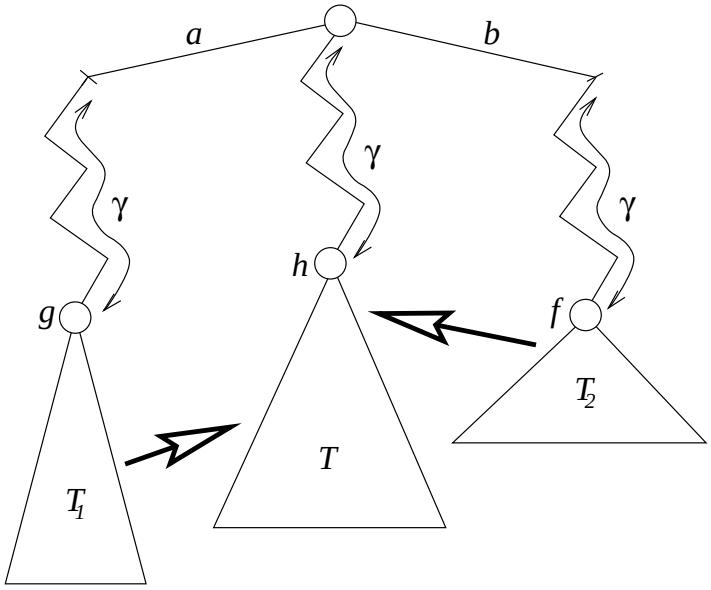

FIG. 16 – T fusion de T_1 et de T_2

Preuve. Puisque $head_{i-1} \neq \epsilon$, posons $head_{i-1} = a.\gamma$ avec $a \in \mathcal{A}$ et $\gamma \in \mathcal{A}^*$. Il existe donc $j-1 < i-1$ tel que $a\gamma$ soit le plus long préfixe commun à $x_{j-1..}$ et $x_{i-1..}$. Donc, γ est le plus long préfixe commun à $x_{j..}$ et $x_{i..}$.

- Si $\gamma = head_i$, alors $locus(\gamma)$ est créé à l'étape i.
- Si $\gamma \neq head_i$, alors γ est préfixe de $head_i$ et $head_i$ est le plus long préfixe commun entre $x_{i..}$ et $x_{k..}$ avec $k < i$. Dès lors, γ est le plus long préfixe commun entre $x_{j..}$ et $x_{k..}$. Le nœud $locus(\gamma)$ existe donc, il a été créé à une étape antérieure. □

7.2.3 L'algorithme

Plaçons-nous après l'étape $i-1$. Nous devons maintenant insérer $x_{i..}$. Pour cela, nous pouvons nous aider des valeurs de tous les $liensuf(g)$ avec g nœud interne autre que la racine et autre que $locus(head_{i-1})$. Pour que ce soit vrai à chaque étape, il est indispensable après insertion de $x_{i..}$ dans l'arbre des suffixes, que l'algorithme donne la valeur adéquate à $liensuf(locus(head_{i-1}))$.

Recherchons maintenant $head_i$. Supposons, dans un premier temps, que ni $locus(head_{i-1})$ ni $pere(locus(head_{i-1}))$ ne soient la racine. Soit $head_{i-1} = a\gamma\delta$ avec $a \in \mathcal{A}; \gamma, \delta \in \mathcal{A}^*$ et $\delta = etiq(arc(pere(locus(head_{i-1})), locus(head_{i-1})))$ (voir figure 17).

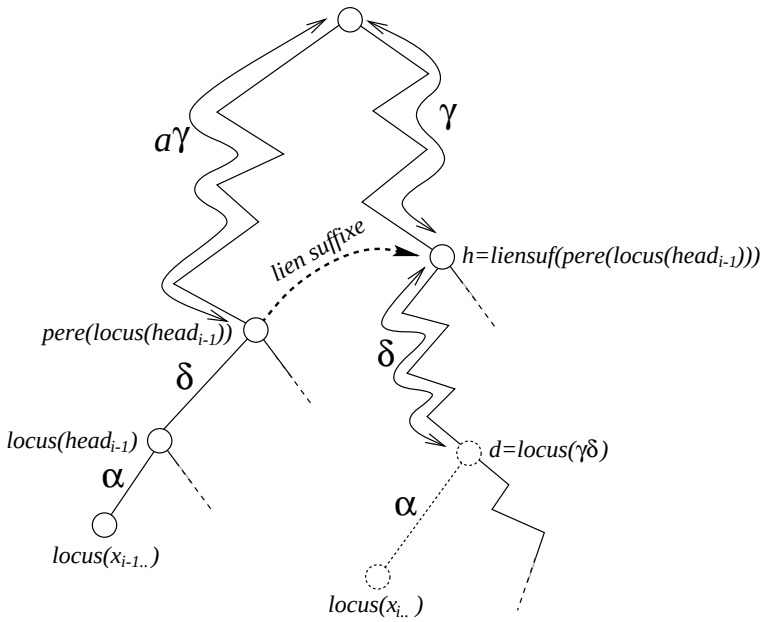

FIG. 17 – Arbre intermédiaire durant l'étape i

Le mot $\gamma\delta$ est préfixe de $x_{i..}$ car $a\gamma\delta$ est préfixe de $x_{i-1..}$. Soit le mot α tel que $x_{i-1..} = a\gamma\delta\alpha$ et $x_{i..} = \gamma\delta\alpha$.

Puisque $fact(pere(locus(head_{i-1}))) = a\gamma$, alors $h = liensuf(pere(locus(head_{i-1})))$ est le locus de γ. Nous savons qu'il existe grâce à la proposition 6. On accède donc au nœud h à partir de $pere(locus(head_{i-1}))$ en temps constant. Une première amélioration par rapport à l'algorithme en force brute consiste à rechercher $head_i$ non plus à partir de la racine mais à partir de h : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(h, \delta\alpha, IndiceTail)$.

Les propositions 7 et 8 permettent d'accélérer également la recherche de $head_i$ à partir du nœud h.

Proposition 7 *Le mot $\gamma\delta$ est préfixe d'un suffixe déjà représenté dans l'arbre actuel.*

Preuve. Puisque $a\gamma\delta = head_{i-1}$, $a\gamma\delta$ est le plus long préfixe commun entre $x_{i-1..}$ et $x_{j..}$ avec $j < i-1$. Donc $\gamma\delta$ est préfixe du suffixe $x_{j+1..}$ inséré à l'étape $j+1 < i$. □

Proposition 8 *Si le locus de $\gamma\delta$ n'existe pas dans l'arbre actuel, alors $head_i = \gamma\delta$.*

Preuve. Puisque $a\gamma\delta$ est le plus long préfixe commun entre $x_{i-1..}$ et $x_{j..}$ avec $j < i-1$, $\gamma\delta$ est le plus long préfixe commun entre $x_{i..}$ et $x_{j+1..}$. Si le locus de $\gamma\delta$ n'existe pas, alors $\gamma\delta$ est le plus long préfixe commun entre $x_{i..}$ et tout mot de l'arbre actuel, en particulier $x_{j+1..}$, ce qui correspond à la définition de $head_i$. □

Si le locus de $\gamma\delta$ existe déjà dans l'arbre, cela signifie que $\gamma\delta$ n'est pas forcément le plus long préfixe commun entre $x_{i..}$ et un mot représenté dans l'arbre actuel (le raisonnement est identique à celui de la remarque 2). Il faut alors rechercher $head_i$ en partant du nœud $d = locus(\gamma\delta)$ en parcourant les caractères de α un à un :

$$locus(head_i) \leftarrow \text{LOCUSPRÉFIXECOMMUN}(d, \alpha, IndiceTail)$$

Puisque $\gamma\delta$ possède un locus étendu dans l'arbre, la recherche (et la création éventuelle) de $d = locus(\gamma\delta)$ à partir du nœud h peut se faire beaucoup plus rapidement qu'en utilisant LOCUSPRÉFIXECOMMUN. Il suffit, à partir de h, de suivre le chemin étiqueté par δ, en ne testant que les premières lettres des étiquettes des arcs. Lorsque tout le mot δ a été lu, si on se trouve au milieu d'un arc, alors COUPEARC est appelé pour créer le nouveau nœud d. Dans le cas contraire, cela signifie que le nœud existe déjà.

Soit RECHERCHERAPIDE la fonction capable d'effectuer une telle recherche. Son algorithme est présenté à la figure 18.

```
RECHERCHERAPIDE(g, w)
 1  NœudCourant ← g
 2  i ← 1                                  /* Position courante dans w */
 3  Trouve ← Faux
 4  TantQue Pas(Trouve)
 5  Faire Si i > |w|                       /* Le locus de w existe */
 6          Alors Retourner NœudCourant
 7          Sinon h ← ACCÈDEFILS(NœudCourant, w_i)
 8                                         /* On accède au fils grâce à la première lettre w_i */
 9                l ← |etiq(arc(NœudCourant, h))|
10                Si l > |w| − i + 1       /* L'arc est trop long */
11                   Alors Trouve ← Vrai
12                   Sinon NœudCourant ← h /* L'arc est trop court */
13                         i ← i + l
14  Retourner COUPEARC(NœudCourant, w_i..)
```

FIG. 18 – Fonction RECHERCHERAPIDE qui recherche (et construit) le locus de w dans l'arbre de racine g

La recherche et construction éventuelle de $d = locus(\gamma\delta)$ se fait donc simplement, à partir du nœud h par l'appel : $d \leftarrow$ RECHERCHERAPIDE(h, δ). Si un nouveau nœud est créé, alors c'est $head_i$: $head_i \leftarrow d$ (proposition 8), sinon il faut rechercher $head_i$ en suivant les caractères de α un

à un à partir de $d : head_i \leftarrow$ LOCUSPRÉFIXECOMMUN$(d, \alpha, IndiceTail)$. Lorsque $locus(head_i)$ est connu, il suffit alors de lui créer le fils $locus(x_{i..})$ comme nouvelle feuille : CRÉENŒUD$(locus(head_i), j, n)$ où $j = IndiceTail + |\gamma\delta| + i - 1$ si LOCUSPRÉFIXECOMMUN a été appelé ou $j = n - |\alpha| + 1$ si LOCUSPRÉFIXECOMMUN n'a pas été appelé.

Dans tous les cas, l'appel à RECHERCHERAPIDE a permis de localiser $d = locus(\gamma\delta)$ qui est la valeur correcte à donner au lien suffixe de $locus(head_{i-1})$.

Il nous reste à comprendre ce qu'il faut faire lorsque $pere(locus(head_{i-1}))$ est la racine ou n'existe pas.

Si $pere(locus(head_{i-1}))$ est la racine, alors la démarche est légèrement différente du cas général puisque le lien suffixe de la racine n'existe pas. Posons $head_{i-1} = a\delta$ et $x_{i..} = \delta\alpha$ avec $a \in \mathcal{A}$ et $\delta, \alpha \in \mathcal{A}^*$. En suivant le même raisonnement que dans le cas général (propositions 7 et 8), δ est préfixe d'un mot qui existe déjà dans l'arbre et si son locus n'existe pas alors $head_i = \delta$. On cherche donc $d \leftarrow$ RECHERCHERAPIDE$(racine, \delta)$. Si d existait déjà alors $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(d, \alpha, IndiceTail)$, autrement $locus(head_i) \leftarrow d$. Il suffit alors, comme dans le cas général, de créer la feuille $locus(x_{i..})$ de père $locus(head_i)$. Le lien suffixe de $locus(head_{i-1})$ est le nœud d.

Si $locus(head_{i-1})$ est la racine, alors $pere(locus(head_{i-1}))$ n'existe pas et nous ne disposons d'aucune information nous permettant de court-circuiter la recherche de $head_i$. On fera alors comme dans l'algorithme en force brute :
$locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(racine, x_{i..}, IndiceTail)$ suivi de la création de la nouvelle feuille $locus(x_{i..})$. Dans ce cas, il ne faut pas créer de lien suffixe pour $locus(head_{i-1})$ car il n'est pas défini pour la racine.

La fonction de construction de l'arbre des suffixes par la méthode de McCreight est présentée à la figure 19. Dans cet algorithme, une structure de donnée d'arbre des suffixes est implicitement connue des fonctions INITARBRE, LOCUSPRÉFIXECOMMUN, RECHERCHERAPIDE et CRÉENŒUD qui la consultent et la modifient. Les $locus(\ldots)$ désignent les nœuds de cette structure implicite et $racine$ désigne sa racine.

Pour illustrer le processus, reprenons l'exemple proposé par McCreight [McC76, Ste94] :
$x = \mathsf{BBBBBABABBBAABBBBB\$} \in \{\mathsf{A}, \mathsf{B}, \$\}^*$.

Plaçons-nous à la fin de l'étape 13, nous devons insérer le suffixe $x_{14..} = \mathsf{BBBBB\$}$ (voir figure 20). En reprenant les notations de l'algorithme, nous avons $\gamma = \mathsf{B}$, $\delta = \mathsf{BB}$ et $\alpha = \mathsf{BB\$}$. Le nœud $h = liensuf(locus(pere(head_{13})))$ est indiqué sur la figure. La recherche $d \leftarrow$ RECHERCHERAPIDE(h, δ) nous amène droit sur un nœud qui existait déjà. La recherche de $head_{14}$ se poursuit donc par : $locus(head_{14}) \leftarrow$ LOCUSPRÉFIXECOMMUN$(d, \alpha, IndiceTail)$ qui construit $locus(head_{14})$ sur l'arc joignant le locus du suffixe $x_{1..}$ à son père. La nouvelle feuille représentant $x_{14..}$ est alors créée.

7.2.4 Étude de complexité de l'algorithme

Nous supposons que l'alphabet est déterminé et contient un petit nombre de lettres. L'accès à un fils d'un nœud se fait donc en temps constant. Si ce n'est pas le cas, les résultats de complexité présentés ici doivent être multipliés par la taille $\#\mathcal{A}$ de l'alphabet ou par $\log \#\mathcal{A}$ selon la structure de données utilisée (voir section 5).

Théorème 1 *L'algorithme de McCreight construit l'arbre des suffixes de* $x = x_1 x_2 \ldots x_n$*, avec* $x_n \neq x_i, \forall i : 1 \leq i < n$*, en temps* $O(n)$.

Preuve. Fixons-nous à l'étape i. Toutes les instructions se font en temps constant excepté les appels à RECHERCHERAPIDE et LOCUSPRÉFIXECOMMUN. Comptabilisons séparément les opérations

```
McCreightST(x, n)
 1  InitArbre()
 2  head_0 ← ε
 3  locus(head_0) ← racine
 4  Pour i ← 1 Jusque n
 5  Faire Si locus(head_{i-1}) = racine
 6          Alors locus(head_i) ← LocusPréfixeCommun(racine, x_{i..}, IndiceTail)
 7                j ← IndiceTail + i − 1
 8          Sinon Si pere(locus(head_{i-1})) = racine
 9                  Alors u ← head_{i-1}
10                        δ ← u_{2..}      /* On supprime la première lettre */
11                        h ← racine
12                        γ ← ε
13                  Sinon δ ← etiq(arc(pere(locus(head_{i-1})), locus(head_{i-1})))
14                        h ← liensuf(pere(locus(head_{i-1})))
15                        γ ← etiq(racine, h)
16                α ← etiq(arc(locus(head_{i-1}), locus(x_{i-1..})))
17                d = RechercheRapide(h, δ)
18                Si d n'a qu'un seul fils
19                  Alors              /* Cas où d vient d'être créé */
20                        locus(head_i) ← d
21                        j ← n − |α| + 1
22                  Sinon locus(head_i) ← LocusPréfixeCommun(d, α, IndiceTail)
23                        j ← IndiceTail + |γδ| + i − 1
24                liensuf(locus(head_{i-1})) ← d
25        CréeNœud(locus(head_i), j, n)
```

FIG. 19 – Algorithme de McCreight

effectuées par RECHERCHERAPIDE et par LOCUSPRÉFIXECOMMUN.

Pour plus de facilités dans les notations, posons $pere_i = fact(pere(locus(head_i)))$.

7.2.4.1 RECHERCHERAPIDE

Il s'agit de rechercher rapidement le locus de $\gamma\delta$ à partir du nœud h (voir figure 17) : $d \leftarrow$ RECHERCHERAPIDE(h, δ).

Le temps de travail de la fonction est proportionnel au nombre de nœuds intermédiaires visités : le temps est en $O(|\delta|)$.

1er cas : h n'est pas la racine, c'est-à-dire que $pere_{i-1} \neq \epsilon$.

Dès lors, écrivons $O(|\delta|) = O(|\gamma\delta| - |\gamma|) = O(|\gamma\delta| - |pere_{i-1}| + 1) = O(|\gamma\delta| - |pere_{i-1}|)$.

A. Si RECHERCHERAPIDE ne crée pas de nouveau nœud, on a $|\gamma\delta| \leq |pere_i|$ et donc le temps de travail de RECHERCHERAPIDE est en $O(|pere_i| - |pere_{i-1}|)$.

B. Si RECHERCHERAPIDE crée un nouveau nœud, alors $\gamma\delta = head_i$ et puisque le temps de travail pour passer de $locus(pere_i)$ à $locus(head_i)$ est constant, le temps de travail de RECHERCHERAPIDE est également en $O(|pere_i| - |pere_{i-1}|)$.

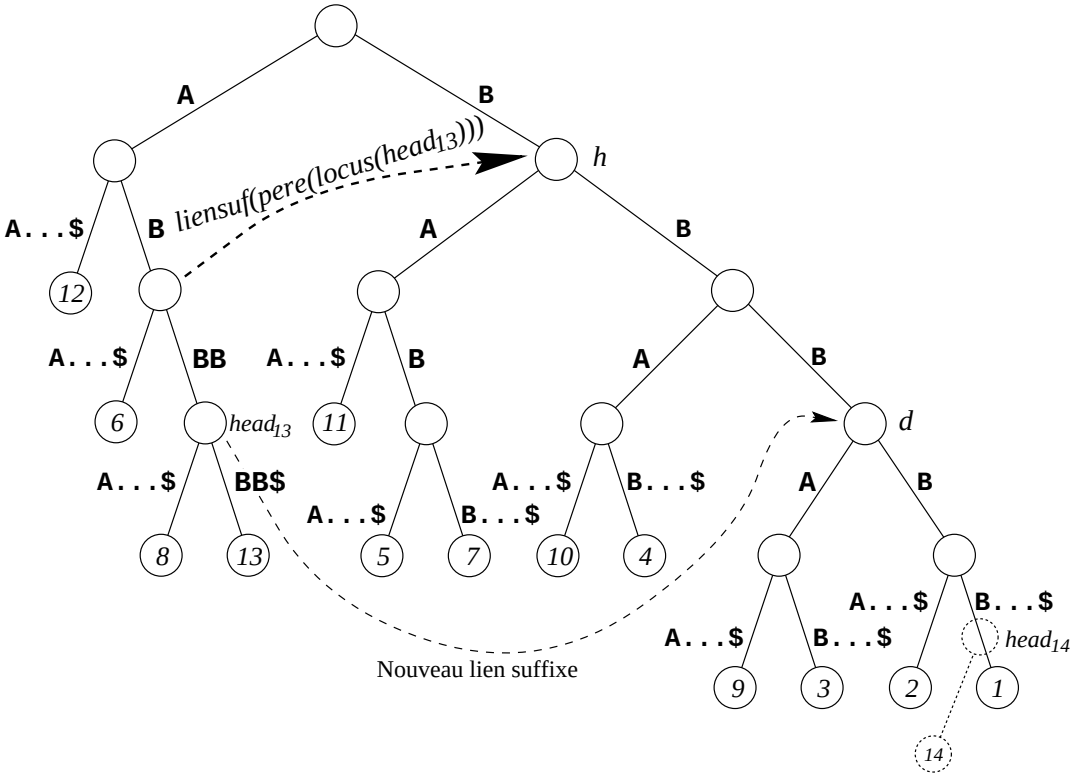

FIG. 20 – Arbre intermédiaire pour $x = \texttt{BBBBBABABBBAABBBBB\$}$ lors de la $14^{\text{ème}}$ étape

$2^{\text{ème}}$ cas : h est la racine. On a donc $|pere_{i-1}| = |\gamma| = 0$ et donc, on peut également écrire que le temps de travail de RECHERCHERAPIDE est en $O(|pere_i| - |pere_{i-1}|)$.

7.2.4.2 LOCUSPRÉFIXECOMMUN

Il s'agit de trouver le locus de $head_i$ lorsque RECHERCHERAPIDE a été appelée pour localiser d ou lorsque $locus(head_{i-1})$ est la racine.

1^{er} cas : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(d, \alpha, IndiceTail)$.

Soit $\alpha = \alpha'.tail_i$. Le temps de travail de LOCUSPRÉFIXECOMMUN est en $O(|\alpha'|)$. Or $|\alpha'| = |\gamma\delta\alpha'| - |a\gamma\delta| + 1 = |head_i| - |head_{i-1}| + 1$ et donc le temps de travail est en $O(|head_i| - |head_{i-1}|)$.

$2^{\text{ème}}$ cas : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(racine, x_{i..}, IndiceTail)$.

Le temps de travail de LOCUSPRÉFIXECOMMUN est en $O(|head_i|) = O(|head_i| - |head_{i-1}|)$ puisque $|head_{i-1}| = 0$.

7.2.4.3 Temps global

Le temps consacré à l'insertion d'un suffixe $x_{i..}$ est donc en $O(|pere_i| - |pere_{i-1}|) + O(|head_i| - |head_{i-1}|) + O(1)$.

Le temps total de l'algorithme de McCreight est en

$$\sum_{i=1}^{n} O(|pere_i| - |pere_{i-1}|) + \sum_{i=1}^{n} O(|head_i| - |head_{i-1}|) + \sum_{i=1}^{n} O(1)$$
$$= O(|pere_n| - |pere_0|) + O(|head_n| - |head_0|) + O(n)$$
$$= O(n)$$

En vertu du fait que $|pere_n| = |pere_0| = |head_n| = |head_0| = 0$. □

Références

[BM77] BOYER, R. S., ET MOORE, J. S. – A fast string searching algorithm. *Comm. ACM*, vol. 20, n° 10, 1977, pp. 762–772.

[CP91] CROCHEMORE, M., ET PERRIN, D. – Two-way string-matching. *J. Assoc. Comput. Mach.*, vol. 38, n° 3, 1991, pp. 651–675.

[CR94] CROCHEMORE, M., ET RYTTER, W. – *Text algorithms*. – Oxford University Press, 1994.

[KMP77] KNUTH, D. E., MORRIS, JR, J. H., ET PRATT, V. R. – Fast pattern matching in strings. *SIAM J. Comput.*, vol. 6, n° 1, 1977, pp. 323–350.

[Knu73] KNUTH, D. E. – *The art of computer programming : Sorting and searching*. – Reading, MA, Addison-Wesley, 1973, volume 3.

[McC76] MCCREIGHT, E. M. – A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, vol. 23, n° 2, 1976, pp. 262–272.

[Mor68] MORRISON, D. R. – PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. Assoc. Comput. Mach.*, vol. 15, 1968, pp. 514–534.

[MR80] MAJSTER, M. E., ET RYSER, A. – Efficient on-line construction and correction of position trees. *SIAM J. Comput.*, vol. 9, n° 4, 1980, pp. 785–807.

[Riv96] RIVALS, É. – *Algorithmes de compression et applications à l'analyse de séquences génétiques*. – Thèse de Doctorat, Université des Sciences et Technologies de Lille, janvier 1996.

[Ste94] STEPHEN, G. A. – *String searching algorithms*. – World Scientific Press, 1994.

[Ukk92] UKKONEN, E. – Constructing suffix trees on-line in linear time. *Dans : Proceedings of the IFIP 12th World Computer Congress*, édité par van Leeuwen, J. pp. 484–492. – Madrid, 1992.

[Ukk95] UKKONEN, E. – On-line construction of suffix trees. *Algorithmica*, vol. 14, n° 3, 1995, pp. 249–260.

[Wei73] WEINER, P. – Linear pattern matching algorithm. *Dans : Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11. – Washington, DC, 1973.