Identification du vocabulaire des carrés en temps linéaire

Rapport remis dans le cadre du cours Algèbre Computationnelle

Émile Nadeau, NADE10059404

28 juillet 2017

1 Introduction et définitions

Un carré, est une mot $\alpha\alpha$ où α est non vide. On s'intéresse ici au repérage des carrés dans un mot $S = s_0 s_1 \cdots s_{n-1}$ donné et de longueur n sur un alphabet Σ de taille finie. On note $S[i] := s_i$ et $S[i : j] := s_i s_{i+1} \cdots sj - 1$.

Définition 1. *Deux carrés $\alpha\alpha$ et $\alpha'\alpha'$ sont de* types *différents si $\alpha \neq \alpha'$.*

Définition 2. *Soit w un mot. Le* vocabulaire des carrés *de w est l'ensemble des types de carré qui apparaisse dans w.*

Pour des raisons de simplicité, on référera simplement au *vocabulaire* du mot. En 1998, Fraenkel et Simpson démontre dans [1] que la taille du vocabulaire est linéaire en la taille du mot. Plus précisément, ils démontrent le théorème suivant :

Théorème 1. *Soit S un mot de longueur n. Pour chaque position i dans S, il y a au plus deux types de carrés dont l'occurrence la plus à droite commence à i. En particulier, a taille du vocabulaire de S est bornée par $2n$.*

La question naturelle qui survient lorsqu'on sait que la taille du vocabulaire est en $\mathcal{O}(n)$ est de savoir s'il est possible de trouver une occurrence du type de chaque carré d'un mot en $\mathcal{O}(n)$. La solution à ce problème est présentée par Gusfield et Stoye pour un alphabet fini de taille fixée dans [3]. Nous présenterons ici leurs résultats et une implémentation de leur algorithme. L'algorithme se décompose en trois phases qui s'effectuent toutes en temps et espace linéaire.

Pour leur algorithme, ils utilisent l'arbre suffixe à la fois comme outil de calcul et comme support pour le résultat de l'algorithme. Commençons donc par quelques rappels sur l'arbre suffixe.

1.1 Arbres suffixes

L'arbre suffixe de S, noté $T(S)$, est une compactification de la trie suffixe de S, cette dernière étant l'automate déterministe en forme d'arbre reconnaissant tous les suffixes de S. L'arbre suffixe peut être construit et stocké en $\mathcal{O}(n)$ comme démontré par Ukkonen dans [5]. Toutefois, avec l'algorithme de construction de Ukkonen et implémenté dans Sage, un suffixe n'est pas forcément associé à une feuille dans T. Pour expliciter tous les états finaux, on ajoute à la fin du mot un symbole \$ qui n'apparaît nul part ailleurs dans le mot. On trouve en Annexe A un exemple d'arbre suffixe.

On dira qu'un nœud v de l'arbre suffixe à une *étiquette de chemin* $L(v)$ si la lecture des arêtes de la racine au noeud v donne le mot $L(v)$. Par exemple, le noeud 5 dans l'exemple de l'Annexe A à une étiquette de chemin $L(5) = abaab$. On note $D(v) = |L(v)|$ la *profondeur de mot* du noeud v. Le noeud 5 dans notre exemple a donc une profondeur de mot $D(5) = 5$.

On peut lire tous les facteurs d'un mot dans l'arbre suffixe car on peut y lire tous les suffixes et un facteur est forcément préfixe d'un certain suffixe. Pour marquer un facteur particulier dans l'arbre suffixe, on enregistre le point où il se termine lorsqu'on le lit à partir de la racine. Pour retrouver ce facteur, il suffit alors de lire les arêtes sur le chemin de la racine au point qui est marqué. Précisément, si la lecture d'un facteur s'arête après avoir lu ℓ lettres sur une arête (u, v) alors on note le point final $((u, v), \ell)$. On note que

$1 \leq \ell \leq D(v) - D(u)$ car si on veut indiquer que la lecture s'arrête à u, on prendra l'arête entrante de u et que le nombre de lettre lues sur (u, v) ne peut pas excéder le nombre à ajouter pour passer de $L(u)$ à $L(v)$.

Le *lien suffixe* d'un noeud v avec étiquette de chemin aw (a une lettre, w un mot) est défini comme $f(v) = v'$ avec v' tel que $L(v') = w$. Le lien suffixe fait partie de la structure de données d'arbre suffixe et est calculé durant la construction. Par la suite, le calcul du lien suffixe d'un noeud ce fait en temps constant.

On a aussi que chaque position dans S est associée avec la feuille du suffixe commençant à cette position. Cette association peut être calculée en temps linéaire par un parcours de l'arbre.

1.2 Relation de couverture

On identifiera une occurrence d'un type de carré par une paire (i, l) où i indique la position de départ du carré dans le mot et l la longueur du carré. Dans S la paire (i, l) indique donc le facteur $s_i s_{i+1} \cdots s_{i+l-1}$. Le vocabulaire peut alors être représenté par un ensemble de paires (i, l) spécifiant chacune une occurrence d'un type différent de carré du vocabulaire. Pour w un mot et a une lettre, on dit que wa est la *rotation à droite* de aw.

Définition 3. *On dit $i, i+1, \ldots, j-1, j$ forme une l-*série *de carrés si (i, l), $(i+1, l)$, …, (j, l) sont tous des carrés.*

Par exemple, dans *aabaabaaa*\$ les positions $0, 1, 2$ forme une 6-série.

Définition 4. *On dit qu'un carré (i, l)* couvre *un carré (j, l) si $i, i+1, \ldots, j$ forme une l-série. On dit qu'un ensemble de paire est un* ensemble couvrant *si au moins une occurrence de chaque carré du vocabulaire est couverte par une paire dans l'ensemble. Si l'occurrence la plus à gauche de chaque carré du vocabulaire est couverte par une paire de l'ensemble couvrant on dira alors que c'est un* ensemble couvrant le plus à gauche.

Si on reprend l'exemple *abaabaabbaaabaaba*\$ on a que $(0, 6)$ couvre $(2, 6)$ car $0, 1, 2$ forme une 6-série. Dans notre exemple, le vocabulaire est $\{aa, bb, aabaab, abaaba, baabaa\}$ donc $\{(0, 6), (7, 2), (10, 2)\}$ forme un ensemble couvrant mais pas le plus à gauche alors que $\{(0, 6), (2, 2), (8, 2)\}$ est un ensemble couvrant le plus à gauche.

2 Algorihtmes préliminaires

2.1 Décomposition de Lempel-Ziv

La *décomposition de Lempel-Ziv* une factorisation d'un mot $S = s_0 s_1 \cdots s_{n-1}$ en facteur $S[i_B : i_{B+1}]$ où i_B est définie récursivement comme $i_1 = 1$ et $i_{B+1} = i_B + \max(1, l_{i_B})$ avec l_k indiquant la longueur du plus long préfixe de $s_k \cdots s_{n-1}$ qui a une occurrence dans S qui commence avant s_k. On appel ces facteurs des *blocs*.

Pour construire la décomposition, on note d'abord que le premier bloc est la première lettre car clairement, $l_0 = 0$. Ensuite, pour trouver le bloc suivant, on considère la position k suivant le dernier bloc construit. Si on a la première apparition d'une lettre alors on a un bloc de longueur 1 (la lettre seulement) car l_k est nul. Si la lettre est déjà apparu alors $l_k \geq 1$ et donc le bloc débutant à cette lettre sera le plus long facteur commençant à cette position qui a une occurrence dans S débutant avant cette position.

On peut construire cette décomposition en temps linéaire à partir de l'arbre suffixe. Pour ce faire, on utilise la propriété que les étiquettes des arcs de l'arbre indique la première apparition de ce facteur. Plus précisément, si on a un facteur du mot dont la lecture s'arrête sur une arête d'étiquette (i, j) alors la dernière lettre de la première apparition de ce facteur est entre i et $j-1$. Pour trouver le plus long facteur commençant à une certaine position dans S et ayant une occurrence commençant avant, on peut donc lire le facteur dans l'arbre suffixe et s'arrêter lorsqu'on dépasse la fin du facteur qu'on est entrain de lire.

Pour clarifier, voici une exemple avec le mot *cacao* (voir Figure 1). On cherche le plus long facteur qui commence à la position 2. À cette position, ce trouve un c on suit donc la c-transition à partir de la racine qui est l'arête $(0, 2)$. On a donc une apparition du facteur *ca* avant celle commençant en position 2. Après *ca* la prochaine lettre est *o*. La *o*-transition à partir l'état 3 est l'arête $(4, 5)$ on s'arrête donc car le premier facteur cao se termine en 4 et est donc celui commençant en 2.

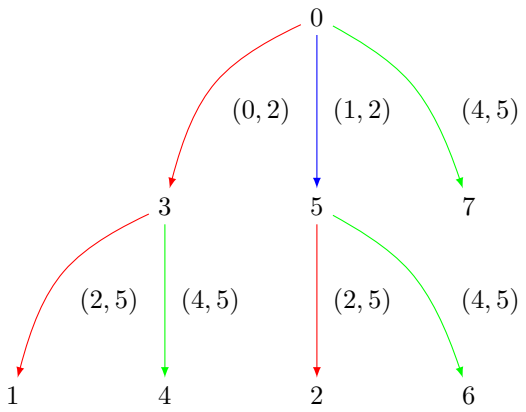


FIGURE 1 – Arbre suffixe de cacao

La méthode détaillée pour calculer la décomposition est décrite dans l'Algorithme 1. On se convainc facilement que cet algorithme a une complexité en $\mathcal{O}(n)$ où n est la longueur du mot. En effet, chaque instruction s'effectue en temps constant (on suppose que la taille de l'alphabet est finie) et les boucles n'ont pour effet que de lire une seule fois S.

Algorithme 1 Décomposition de Lempel-Ziv

```
1:  fonction LZ-DECOMPOSITION(T : arbre suffixe) : liste d'entier
2:      blocs ← [0]
3:      i ← 0
4:      S ← T.MOT()
5:      tant que i < |S| faire
6:          l ← 0
7:          Soit (x, y) la S[i]-transition à partir de la racine de T
8:          tant que x < i + l faire
9:              si y = |S| alors
10:                 l ← |S| − i
11:             sinon
12:                 l ← l + y − x
13:             fin si
14:             si i + l ≥ |S| alors
15:                 l ← |S| − i
16:                 arrêt
17:             fin si
18:             Soit (x, y) la S[i + l]-transition à partir de l'extrémité de (x, y)
19:         fin tant que
20:         blocs.AJOUTER(max(l, 1))
21:     fin tant que
22: fin fonction
```

2.2 Plus longues extensions communes vers l'avant et l'arrière

Un autre sous-problème à résoudre est le calcul des plus longues extensions vers l'avant et vers l'arrière dans un mot. Ce problème consiste, étant donné deux positions i et j dans S, à trouver le plus long facteur commun commençant (resp. terminant) à i et j. On nomme ce facteur *plus longue extension commune vers l'avant* (resp. *vers l'arrière*). Par exemple dans le mot *aabaabaa* pour les position 1 et 4 la plus longue extension commune est *abaa* vers l'avant et *aa* vers l'arrière.

On présente ici le traitement du problème de plus longue extension commune vers l'avant puisque pour celle vers l'arrière il suffit de considérer le mot à l'envers. Dans le chapitre de 8 de [2], Gusfield présente un algorithme qui permet après un prétraitement en $\mathcal{O}(n)$ de retrouver le plus petit ancêtre commun de n'importe quelle paire de nœud dans un arbre en temps constant.

Dans notre cas précis, le prétraitement s'effectue en deux partie On doit prétraiter l'arbre suffixe pour le calcul du plus petit ancêtre commun. Par un simple parcours, on crée aussi une association qui permet en temps constant de trouver $D(v)$ pour tout nœud v. L'Algorithme 2 permet alors de retrouver en temps constant la longueur de la plus longue extension commune en temps constant.

Algorithme 2 Calcul de la plus longue extension commune

```
1: fonction LCE(i,j : position) : entier
2:     v_1 ← la feuille associée au suffixe commençant en position i
3:     v_2 ← la feuille associée au suffixe commençant en position j
4:     v ← plusPetitAncêtreCommun(v_1, v_2)
5:     retourner D(v)
6: fin fonction
```

Pour l'implémentation, j'ai toutefois choisi l'approche naïve qui consiste à comparer caractère par caractère les chaînes démarrant aux positions i et j jusqu'à trouver une différence (voir Algorithme 3)

Algorithme 3 Calcul de la plus longue extension commune (approche naïve)

```
1: fonction LCE(S :mots, i, j : positions) : entier
2:     l ← 0
3:     tant que S[i + l] = S[j + l] faire
4:         l ← l + 1
5:     fin tant queretourner l
6: fin fonction
```

Le choix de cette approche plus simple est justifié par les résultats de Ilie, Navarro et Tinta qui dans [4] démontre que l'approche naïve est plus efficace en moyenne et plus rapide en pratique. L'intérêt de cette méthode est aussi sa simplicité d'implémentation.

2.3 Stratégie compter et sauter

Dans un arbre suffixe, la stratégie compter et sauter est une façon accélérée de lire un facteur à partir d'un nœud si on sait à l'avance qu'on peut lire ce facteur à partir de ce noeud. La stratégie est la suivante, si on veut lire le facteur $s_i s_{i+1} \cdots s_{j-1}$ à partir du noeud u. On regarde la s_i transition à partir de u et si le facteur associé à l'arête est moins long que la longueur du facteur à lire, on saute l'arête et on répète avec le prochain noeud et le reste du facteur à lire. Si on veut lire le facteur $s_i s_{i+1} \cdots s_{j-1}$ dans $T(S)$ à partir du noeud u, la façon de faire est alors décrite par l'Algorithme 4.

Algorithme 4 Stratégie compter et sauter

```
1: fonction compterEtSauter(u : noeud, i, j : positions) : entier
2:     Soit (u, v) l'arête de la s_i-transition de u
3:     l ← D(v) − D(u)
4:     si l ≥ j − i alors
5:         retourner ((u, v), j − i)
6:     fin si
7:     retourner CompterEtSauter(v, i + l, j)
8: fin fonction
```

Cette stratégie permet de sauter chaque arête en temps constant et d'avoir une complexité au pire cas en $O(j-i)$ alors que tous les cas on une complexité en $\Theta(j-i)$ si on lit le facteur au complet.

3 Phase I : Trouver un ensemble couvrant le plus à gauche

L'objectif de cette phase est de construire en temps linéaire un ensemble couvrant le plus à gauche comme décrit dans la Section 1.2. Pour ce faire, on utilisera la décomposition de Lempel-Ziv. On commence donc par quelques résultats sur le liens entre les positions des occurrences les plus à gauche des carrés et les blocs.

Lemme 1. *La seconde moitié d'un carré $\alpha\alpha$ d'un mot S doit toucher au plus deux blocs de la décomposition de Lempel-Ziv de S.*

Démonstration. Supposons que la seconde moitié α touche plus de deux bloc. Soit β le premier bloc complètement inclus dans α et η le suffixe de α suivant β. Alors η est forcément non-vide car le α de droite touche au moins trois blocs. Comme $\alpha\alpha$ est un carré, il y a une occurrence précédente de $\beta\eta$ dans S ce qui contredit que le bloc β est de longueur maximale. □

Lemme 2. *L'occurence la plus à gauche d'un carré dans S touche au moins deux blocs de la décomposition de Lempel-Ziv de S.*

Démonstration. Supposons que la première apparition d'un carré apparaisse complètement dans un bloc β. Alors par définition de la décomposition, β a une occurrence plus à gauche dans le mot et donc le carré aussi. □

On appel *centre* d'un carré $\alpha\alpha$ la dernière lettre de la première moitié. On alors le théorème suivant où le bloc B est le bloc $S[i_B : i_{B+1}]$.

Théorème 2. *Si la première occurrence (la plus à gauche) d'un carré $\alpha\alpha$ a son centre dans un bloc B alors soit*

- *(Condition 1) $\alpha\alpha$ a son début à l'intérieur du bloc B et sa fin dans le bloc $B+1$;*
  ou
- *(Condition 2) $\alpha\alpha$ a son début dans le bloc $B-1$ ou avant et sa fin dans le bloc B ou $B+1$.*

Démonstration. Soit $\alpha\alpha$ un carré ayant son centre dans le bloc B. Supposons que $\alpha\alpha$ ne satisfassent pas la condition 1. Alors ou bien son début est avant le bloc B ou bien sa fin n'est pas dans le bloc $B+1$.

Si son début n'est pas dans le bloc B alors il est dans le bloc $B-1$ ou avant. Si son centre est le dernier caractère du bloc B alors le bloc $B+1$ a α comme préfixe car on sait que α a une occurrence plus à droite dans la chaîne. On a donc que le dernier caractère du carré est dans $B+1$. Si le centre n'est pas le dernier caractère de B alors la fin du carré est dans $B+1$ car la seconde moitié ne peut toucher plus de deux blocs (Lemme 1).

Si sa fin n'est pas dans le bloc $B+1$, elle doit être dans le bloc B. En effet, le centre ne peut pas être la dernière lettre du bloc B car alors le bloc $B+1$ aurait α comme préfixe et la fin de l'occurrence serait donc dans $B+1$. La fin doit donc être dans B car la seconde moitié d'un carré touche B et ne peut pas toucher plus de deux bloc (Lemme 1). De plus, la première occurrence d'un carré doit toucher au moins deux blocs par le Lemme 2 et donc le début doit être dans le bloc $B-1$ ou avant. □

On dit qu'un carré est de *type 1* (resp. *type 2*) si il satisfait la condition 1 (resp. condition 2) du théorème. Pour chacune des conditions du théorème précédent, on présente ici l'Algorithme 5 qui maintient l'invariant suivant : Après avoir traité les blocs $1, 2, 3, ..., B$ toutes les occurrences les plus à gauches de carrés qui ont leur centre dans les blocs $1, 2, 3, ..., B$ sont couvertes par une des paires engendrées par l'algorithme. Pour pouvoir compléter la preuve des résultat de [3], il faut toutefois noter qu'il y a trois petites erreurs à corriger dans l'algorithme. Pour le type 1, il faut ajouter la condition $start \geq h$ pour le si. Pour le type 2, il faut prendre k seulement à partir de 2 et changer $start + k < h_1$ pour $start + k \leq h_1$

Lemme 3. *Toutes les paires retournées par l'Algorithme 5 représentent des carrés.*

Démonstration. Regardons d'abord le type 1. Si une paire est retournée alors $k_1 + k_2 \geq k$ et on peut décomposer le mot comme illustré à la Figure 2. On a alors que $q - k_2$ indique le début du premier facteur α et donc $(q - k_2, 2k)$ est bien un carré puisqu'il indique $(tuv)^2$. Si $q - k + 1$ est plus grand que $q - k_2$ alors $k_2 \geq k$ et la définition de k_2 nous donne donc

$$S[q : h_1] = S[h_1 - k : h_1] = S[q - k : q]$$

Algorithme 5 Traitement du bloc B pour l'ensemble couvrant le plus à gauche

Soit h le point de départ de B
Soit h_1 le point de départ de $B+1$
fonction CARRÉSTYPE1( ) : paires de carrés
    pour $k = 1, 2..., |B|$ **faire**
        $q \leftarrow h_1 - k$
        $k_1 \leftarrow$ longueur de la plus longue extension commune vers l'avant commencant à h_1 et q
        $k_2 \leftarrow$ longueur de la plus longue extension commune vers l'arrière commencant à $h_1 - 1$ et $q - 1$
        start$\leftarrow \max(q - k_2, q - k + 1)$
        si $(k_1 + k_2 \geq k) \wedge (k_1 > 0) \wedge (start \geq h)$ **alors**
            engendrer (start,$2k$)
        fin si
    fin pour
fin fonction

fonction CARRÉSTYPE2( ) : paires de carrés
    Soit B
    pour $k = 2, 3..., |B| + |B+1|$ **faire**
        $q \leftarrow h + k$
        $k_1 \leftarrow$ longueur de la plus longue extension commune vers l'avant commencant à h et q
        $k_2 \leftarrow$ longueur de la plus longue extension commune vers l'arrière commencant à $h_1 - 1$ et $q - 1$
        start$\leftarrow \max(h - k_2, h - k + 1)$
        si $(k_1 + k_2 \geq k) \wedge (k_1 > 0) \wedge (start + k \leq h_1) \wedge (k_2 > 0)$ **alors**
            engendrer (start,$2k$)
        fin si
    fin pour
fin fonction

De plus, si une paire est retournée $k_1 > 0$ donc $S[h_1] = S[q]$. Ainsi, $S[q+1 : h_1+1] = S[q-k+1 : q+1]$ Donc $(q-k+1, 2k)$ est le carré $S[q-k+1 : h_1+1]$.

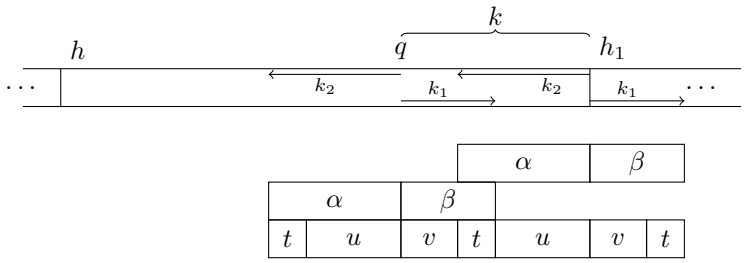


FIGURE 2 – Algorithme 5 pour le type 1 avec $k_1 + k_2 \geq k$

Dans le cas du type 2 la preuve est similaire au type 1 mais h joue le rôle qu'avait q et q le rôle qu'avait h_1 (voir Figure 3). □

Lemme 4. *Si S possède deux carrés $(i, 2k)$ et $(j, 2k)$ tel que $i \leq j \leq i + k$ alors $(i, 2k)$ couvre $(j, 2k)$.*

Démonstration. Soit $\alpha\alpha$ et $\beta\beta$ ces carrés. Par les conditions sur j, les deux carrés ont une portion commune de longueur au moins k. On peut décomposer la portion couverte par $\alpha\alpha$ et $\beta\beta$ en facteur u et v comme à la Figure 4 de sorte que $|u| + |v| = k$. Pour montrer que $(i, 2k)$ couvre $(j, 2k)$, il faut voir qu'on a une $2k$-série qui va de $i, i+1, \ldots j$. Pour ce faire, on doit voir que $(r, 2k)$ est un carré pour tout $i \leq r \leq j$. Soit u_1 le

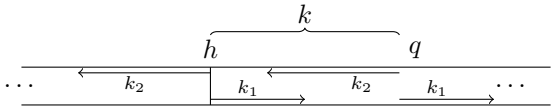


FIGURE 3 – Algorithme 5 pour le type 2 avec $k_1 + k_2 \geq k$

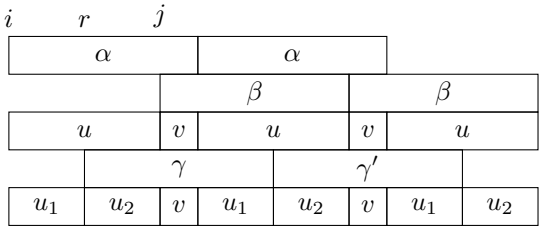


FIGURE 4 – $(i, 2k)$ couvre $(j, 2k)$

préfixe de longueur $r - i$ de u et u_2 le suffixe correspondant. On a alors que $(r, 2k)$ est le mot $(u_2vu_1)(u_2vu_1)$ qui est un carré. □

Proposition 1. *Soit $\gamma\gamma$ un carré de longueur $2k$ de w qui satisfait la condition 1 ou la condition 2. Alors $\gamma\gamma$ est couvert par une des paires générées par l'Algorithme 5.*

Démonstration. Soit c le centre de $\gamma\gamma$. Le carré est donc décrit par la paire $(c - k + 1, 2k)$. Soit B le bloc contenant c et h le point de départ du bloc B. Soit h_1 le départ de $B + 1$.

Si $(c-k+1, 2k)$ satisfait la condition 2 alors sa fin est dans B ou $B+1$ et donc $k \leq |B| + |B+1|$. De plus, son début est dans $B - 1$ ou avant donc $k \geq 2$. Le carré devrait donc être couvert par CARRÉTYPE2 sur le bloc B avec k. Comme son début est dans $B-1$ ou avant et son centre dans B, on a que $c-k+1 < h$ et $h \leq c$ donc $h \leq c < h+k-1$. Comme $q = h+k$ on a alors $c < q-1$ et $c \geq h$ donc $c+1 < q \leq c+k$. Donc q indique une position dans le second facteur γ. En particulier, notons que h et $q = h + k$ indique la même position dans les deux facteurs de $\gamma\gamma$ (voir Figure 5). Donc si h indique la i^{e} lettre de γ, on a que $k_1 \geq |\gamma| - i$ et $k_2 \geq i$. Donc $k_1 + k_2 \geq k$. De plus, comme $c < q - 1$, i indique au moins la deuxième lettre du second facteur γ donc $k_2 > 0$. De même, $k_1 > 0$ car on lit au moins la dernière lettre des facteurs γ. Finalement, $(h - k + 1) + k = h + 1 \leq h_1$ et $h - k_2 + k \leq (c - k + 1) + k \leq c + 1 \leq h_1$ (car $h - k_2$ indique la position de départ γ^2 ou une position plus petite) donc $start \leq h_1$.

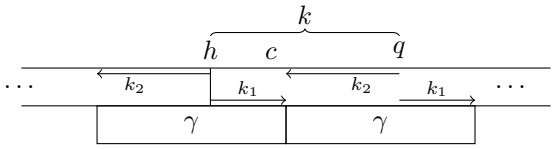


FIGURE 5 – h et q indique les mêmes position dans les deux facteurs de $\gamma\gamma$

Toute les conditions sont donc satisfaites pour qu'une paire soit retournée. Il faut donc voir qu'alors $(start, 2k)$ couvre $(c - k + 1, 2k)$. On sait que $h - k_2 < c - k + 1$ et $h \leq c$, on a donc $start \leq c - k + 1$. De plus, on a vu que $c - k + 1 < h$ et donc $c - k + 1 \leq h + 1 = h - k + 1 + k \leq start + k$. On a donc que

$$start \leq c - k + 1 \leq start + k$$

et donc par le Lemme 4, la paire retournée couvre $(c - k + 1, 2k)$.

Si $(c-k+1, 2k)$ **satisfait la condition 1** alors son début est dans B et sa fin dans $B+1$. Comme son centre est dans B, toute sa première moitié est dans B et donc $k \leq |B|$. Ce carré devrait donc être couvert par carréType1 sur le bloc B avec k.

Soit $q = h_1 - k$. Comme la fin est dans $B+1$, on a que $c + k \geq h_1$ et donc $c \geq q$. Aussi, $c - k < q$ car $c < h_1 = q + k$. En somme, on a $c - k < q \leq c$ et donc q indique un position dans la première moitié de $\gamma\gamma$. De plus h_1 et q indique la même position dans les deux facteur γ (voir Figure 6). De même que dans le cas précédent pour la condition 2, on obtient que $k_1 + k_2 \leq k$ et $k_1 > 0$. Pour qu'une paire soit retournée, il ne manque plus que d'avoir $start \geq h$ mais nous y reviendrons. Supposons que cette dernière condition est

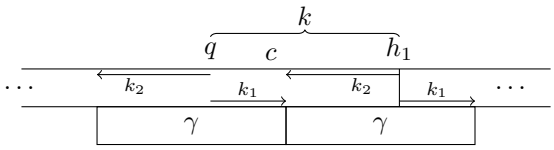


Figure 6 – h_1 et q indique les mêmes position dans les deux facteurs de γ^2

vérifiée et regardons si $(start, 2k)$ couvre $(c-k+1, 2k)$. On a que $q \leq c$ et $q - k_2 \leq q - k + 1$ car $q - k_2$ indique la position de départ γ^2 ou une position plus petite. On a donc $start \leq c - k + 1$. De plus,

$$c - k + 1 \leq q + 1 = h_1 - k + 1 \leq start + k$$

et donc par le Lemme 4, la paire retournée couvre $(c-k+1, 2k)$.

Si on n'a pas que $start \geq h$ aucune paire n'est retournée. Par contre, on a alors que le début de $(start, 2k)$ est dans $B-1$ et son centre est dans B car $q \leq start \leq c$ (car $start \geq q - k + 1$ et le carré couvre le carré de centre c et donc est avant). De plus, sa fin est dans B ou $B+1$ car elle est avant la fin de $(c-k+1, 2k)$ qui est dans $B+1$ par hypothèse. Ainsi $(start, 2k)$ satisfait la condition 2 pour le bloc B et donc a été couverte par une paire retournée par carréType2. Comme la couverture est transitive, on n'a pas à retourner de paire car $\gamma\gamma$ est déjà couvert par une paire retournée pour le type 2. □

Théorème 3. *Lorsqu'on applique l'Algorithme 5 sur tous les blocs, on obtient un ensemble couvrant le plus à gauche en* $\mathcal{O}(|S|)$.

Démonstration. On a d'abord que l'occurrence la plus à gauche de chaque type de carré est couverte. En effet, par le Théorème 2 elle doit satisfaire la condition 1 ou 2 et par la Proposition 1 tous les carrés satisfaisant ces conditions sont couverts.

De plus ce calcul est effectué en temps linéaire. En effet, après un prétraitement linéaire, les calculs de plus longues extensions peuvent être effectués en temps constant. Ce qui détermine la complexité est donc le nombre de tour dans la boucle. Pour carrésType1 la complexité pour un bloc B est donc $\mathcal{O}(|B|)$ alors que pour carrésType2 c'est $\mathcal{O}(|B| + |B+1|)$. La complexité totale est donc en

$$\mathcal{O}\left(\sum |B|\right) = \mathcal{O}(|S|)$$

. □

Pour chaque position i dans le mot, on note $P(i)$ la liste des paires (i, l^i_j) retournées par l'Algorithme 5. Soit $P = \cup_i P(i)$.

Lemme 5. *Sans changer la complexité au pire cas, on peut supposer qu'on obtient les listes* $P(i)$ *avec les* $l^i_j > l^i_{j+1}$ *pour tout* j.

Démonstration. Montrons d'abord que le centre des paires retournées lorsqu'on fait le traitement sur un bloc B ont toutes leur centre dans ce bloc. La condition $start + k \leq h_1$ indique que le centre est avant $B+1$.

Cette condition est requise dans carrésType2 pour générer une paire. Pour carrésType1, on a que

$$\begin{aligned} start + k &= q + k - \min(k_2, k-1) \\ &\leq h_1 - k_1 + k - \min(k_2, k_1 + k_2 - 1) \text{ car } k_1 + k_2 \geq k \\ &= h_1 - k_1 + k - k_2 - \min(0, k_1 - 1) \\ &\leq h_1 - k_1 + k - k_2 \text{ car } k_1 > 0 \\ &\leq h_1 \text{ car } k_1 + k_2 \geq k \end{aligned}$$

On a donc bien que le centre de toutes les paires retournées lors du traitement du bloc B ont leur centre avant $B+1$. On sait aussi que $start + k > h$ pour carrésType1 car $start \geq h$ est une condition requise pour engendrer une paire. Pour carrésType2, on sait que $start + k \geq h + 1 > h$. On a donc toujours $start + k > h$ et donc le centre est bien dans B.

On vient de voir que la position de départ des paires générées se trouve dans le bloc B pour les paires retournées par carrésType1. Pour les paires engendrées par carrésType2, l'indice de la position de départ est avant le bloc B. En effet, $h - k_2 < h$ et $h - k + 1 < h$ car $k > 2$.

On engendre l'ensemble couvrant le plus à gauche en appliquant carréType1 et carréType2 sur le premier bloc, puis le second bloc et ainsi suite. On vérifit que si une paire (i, l) est générée alors toutes les paires (i, l') générées précédemment satisfont $l' < l$. Soit (i, l) une paire retournée lors du traitement du bloc B. Alors toutes les paires générées pour les blocs précédents ont un paramètre de longueur $l' < l$ car leur centre doit être dans les blocs précédents. De plus lorsqu'on traite une bloc B, carréType1 et carréType2 génèrent des paires avec des i différent car l'indice position de départ des paires retournées se trouve dans le bloc B pour le type 1 alors que pour les paires retournées pour le type 2 l'indice de la position de départ est avant le bloc B. Finalement, pour un type particulier sur un bloc particulier, les paires sont retournées en ordre croissant de longueur. On a donc bien que lorsqu'une paire (i, l) est retournée, toutes les paires (i, l') générées précédemment satisfont $l' < l$.

Pour obtenir les listes $P(i)$ comme voulue, il suffit donc de traiter les blocs dans l'ordre décrit précédemment et de d'ajouter en fin de liste $P(i)$ chaque paire générée. Pour avoir l'ordre désiré, on n'a plus qu'à renverser l'ordre de la liste. Ceci ne modifie pas la complexité au pire cas qui reste linéaire. □

4 Phase II : Marquage partiel de l'arbre

Une fois que qu'on a un ensemble couvrant le plus à gauche, on marque certains des carrés obtenus dans l'arbre suffixe avec l'algorithme suivant. On attribut d'abord à chaque feuille de l'arbre l'étiquette i qui indique la position de départ du suffixe associé à cette feuille. On attribue ensuite à chaque enfant la liste $P(i)$ associée à son étiquette. On étiquette aussi chaque noeud interne par la plus petite étiquette de ses enfants. On effectue par la suite un parcours en profondeur de l'arbre et on effectue pour chaque noeud n avec parent p le traitement décrit par l'Algorithme 6 puis après avoir traité tous les enfants, on associe au parent la portion de la liste de l'enfant qui a la même étiquette que lui.

Algorithme 6 Traitement d'une noeud

tant que $P(n)$ n'est pas vide et $D(p) >$ longueur de la paire de tête de $P(n)$ **faire**
  Soit (i, l) la tête de $P(n)$.
  Enregistrer $l - D(p)$ sur l'arc (p, n)
  Retirer (i, l) de $P(n)$
fin tant que

Proposition 2. *La phase de marquage partielle s'effectue en $\mathcal{O}(n)$.*

Démonstration. Par un parcours en largeur de l'arbre, on peut construire un dictionnaire $D(v)$. Par un parcours en profondeur, on procède ensuite à l'étiquetage des nœuds. Finalement un dernier parcours en profondeur permet de traiter les nœuds et d'associer les listes au nœud interne au même moment.

On voit facilement que les deux premiers parcours s'effectue en temps linéaire. Pour le troisième, on utilise une astuce pour s'assurer qu'il soit en temps linéaire. Au lieu d'associer à chaque noeud une liste,

on lui associe une paire (i, pos) où i indique la portion finale de quelle $P(i)$ est associé à ce sommet et pos indique le début de cette portion. La liste peu alors être associé en temps constant. De plus chaque élément de P n'est marqué qu'a une occasion sur l'arbre. Le dernier parcours s'effectue donc en

$$\mathcal{O}(\text{nombre de noeud} + |P|) = \mathcal{O}(n)$$

car l'ensemble couvrant le plus à gauche est de taille linéaire par rapport à la longueur du mot. L'ensemble du marquage partiel peut donc être effectué en $\mathcal{O}(n)$. □

On conclu cette section avec quelque notations. Rappelons que P est l'ensemble des paire retournées par l'Algorithme 5. On note P' l'ensemble des paires de P qui ont été retiré des liste $P(i)$ durant l'Algorithme 6 (*i.e* les paires qui ont été marquée). On note Q l'ensemble des types de carrés donnée par P et Q' le sous-ensemble de Q qui consiste en tous les types de carrés spécifiés par P'.

5 Phase III : Marquage complet de l'arbre

L'objectif de cette phase est de marquer de tous les types de carrés de Q à partir du marquage de la des carrés obtenus à la phase II, *i.e.* les carrés de Q'. Pour ce faire, on utilisera la marche suffixe. Une *marche suffixe* consiste à passer de l'état du mot aw avec a une lettre à l'état du mot w (même si l'état de départ n'est pas explicite) en utilisant le lien suffixe. Soit un état stocké comme $((u, v), l)$ où (u, v) est une arête et l la profondeur de l'état sur cette arête. On procède de la façon suivant pour la marche suffixe. On décompose $aw = a\gamma\beta$ où $L(u) = a\gamma$. On suit le lien suffixe de u pour trouver $f(u)$, l'état associé au facteur γ. On lit ensuite β à partir de $f(u)$ pour trouver l'état associé à $w = \gamma\beta$. Comme on sait qu'on peut lire β à partir de $f(u)$, on utilise la stratégie compter et sauter décrite à la Section 2.3 pour lire β plus rapidement.

Soit $((u', v'), l')$ l'état obtenu par la marche suffixe que l'on vient de décrire. Si à partir de cet état on peut lire la lettre a alors on dit que la marche suffixe est *fructueuse*. Sinon on dit qu'elle est *non fructueuse*. Une marche fructueuse à partie de aw indique donc que la rotation à droite de ce facteur est aussi un facteur de S. Ainsi, une marche fructueuse à partir d'un carré aw indique de le carré wa apparaît aussi dans le mot. Une l-série dans un mot, induit donc une chaîne de marche suffixe fructueuse dans l'arbre suffixe. On introduit maintenant une définition qui permettra de justifier la construction de Q' dans la section précédente.

Définition 5. *Un sous-ensemble du vocabulaire des carrés est dit* suffisant *si chaque élément du vocabulaire peut être atteint à partir d'un élément du sous-ensemble par une chaîne de marche suffixe fructueuse.*

Théorème 4. *Q' est suffisant.*

Démonstration. On remarque d'abord que Q est suffisant. En effet, Q est un ensemble couvrant le plus à gauche donc chaque type de carrés présent dans le mot est couvert par un carré de Q. On peut donc atteindre tous les types de carré par une chaîne de marches suffixes fructueuses car on peut les atteindre par une l-série débutant par un élément de Q.

Par transitivité, pour montrer que Q' est suffisant, on montre que chaque élément de Q est atteignable par une chaîne de marches suffixes fructueuses à partir d'un élément de Q'. Soit Q'' l'ensemble des éléments de Q qui ne peuvent pas être atteint à partir des éléments de Q'. Supposons que Q'' soit non-vide et posons P'' le sous-ensemble des paires de P qui spécifient les carrés de Q''. Soit (j, l) une paire de P'' tel que j soit minimal. Soit $\gamma\gamma$ le carré spécifié par (j, l).

Si (j, l) indiquait la première apparition de $\gamma\gamma$ alors (j, l) ne pourrait pas être dans P''. En effet, on aurait alors que le plus long suffixe du mot qui commence par $\gamma\gamma$ commence à la position j. Donc pour chaque feuille sous l'état associé au facteur $\gamma\gamma$, le suffixe associé ne peut pas commencer avant j donc la liste $P(j)$ doit remonter jusqu'à l'état de $\gamma\gamma$ et donc la paire (j, l) crée une marque de le marquage partiel. On doit donc avoir $(j, l) \in P'$, $\gamma\gamma \in Q'$, $\gamma\gamma \notin Q''$ et donc $(j, l) \notin P''$. Donc (j, l) n'indique pas la première apparition de $\gamma\gamma$.

Soit (i, l) la paire spécifiant la première apparition de $\gamma\gamma$. Comme P est un ensemble couvrant le plus à gauche la paire (i, l) doit être couverte par une paire (r, l) dans P. On a alors $r \leq i < j$. Comme on a choisi la paire (j, l) dans P'' pour que j soit minimal on a que (r, l) n'est pas dans P''. On peut donc atteindre le carré spécifié par (r, l) à partir d'un élément de Q' et (j, l) à partir de (r, l). On a donc que $\gamma\gamma$ n'est pas dans Q'' ce qui est une contradiction. Donc Q'' est vide et Q' est suffisant. □

Pour construire un marquage complet du vocabulaire des carrés, on procède donc comme suit :

1. On effectue un parcours de l'arbre
2. À chaque fois qu'on rencontre un état du marquage partiel, on l'ajoute au marquage complet et on démarre une chaîne de marche suffixe.
3. Tant que les marches suffixes sont fructueuses et qu'on n'arrive pas à un état du marquage partiel, on continue la chaîne et on marque les états visités dans le marquage complet.

Proposition 3. *Avec la procédure décrite précédemment, aucun point n'est marqué plus d'une fois.*

Démonstration. Supposons qu'en partant de deux carrés $\alpha\alpha$ et $\beta\beta$ dans Q' on tombe sur le même carré $\gamma\gamma$. Alors une suite de n rotations (resp. m rotation) transforme $\alpha\alpha$ (resp. $\beta\beta$) en $\gamma\gamma$. Sans perte de généralité, supposons que $m \geq n$. Alors si on applique $m - n$ rotations à droite à $\beta\beta$ on trouve $\alpha\alpha$. Si $m > n$ alors la chaîne de marche suffixe démarrant à $\beta\beta$ se termine à $\alpha\alpha$ car ce dernier carré est dans Q'. La chaîne débutant ne peut donc pas atteindre $\gamma\gamma$. On a donc que $m = n$ et alors on doit avoir $\alpha = \beta$.

Ainsi chaque carré du vocabulaire n'est marqué que par une chaîne partant d'un seul élément de Q' et donc une seule fois. □

En combinant, la Proposition 3 et le Théorème 4 on peut démontrer que le marquage est bien correct.

Théorème 5. *La phase III marque chaque type de carré de S une et une seule fois dans dans $T(S)$.*

On vérifie maintenant que cette dernière phase a bien une complexité dans $\mathcal{O}(n)$. Pour ce faire on débute avec deux lemmes.

Lemme 6. *Il y a au plus deux types de carrés qui se termine sur un arête (u, v),* i.e *qui sont spécifiés sous la forme $((u, v), l)$ avec $l > 0$.*

Démonstration. Soit $R(v)$ la plus grande étiquette d'une feuille dans le sous-arbre induit par le nœud v. On a alors que l'occurrence la plus à droite d'un facteur dont la lecture se termine sur l'arête (u, v) commence à la position $R(v)$. Par le Théorème 1, il y a au plus deux types de carrés dont l'occurrence la plus à droite commence à la position $R(v)$. Donc au plus deux carrés peuvent avoir leur point final sur l'arête (u, v). □

Pour le prochain lemme, nous utiliserons la notion de *type* de lien suffixe. Si l'état u est l'état d'un mot aw avec a une lettre, on dira de $f(u) = v$ est un lien suffixe de type a.

Lemme 7. *Durant la phase de marquage complet, le nombre d'arêtes traversées par l'ensemble des marches suffixes est en $\mathcal{O}(|\Sigma|n)$*

Démonstration. Pour la démonstration, on fixe a une lettre. On démontrera les 5 points suivants :

1. Une marche suivant un lien suffixe de type a ne peut pas sauter une arête terminant par un état ayant un lien suffixe entrant de type a.
2. Soit (u, v) une arête et e une arête sur le chemin de $f(u)$ à $f(v)$. Alors si $f(u)$ est de type a, toutes marches suffixes qui commencent avec un lien suffixe de type a et qui saute e commence sur l'arête (u, v).
3. Chaque arête e est sur au plus un chemin dont les seuls nœuds ayant un lien suffixe de type a entrant sont les extrémités.
4. Une arête e est sautée au plus 2 fois par une marche suivant un lien suffixe de type a
5. Le nombre d'arêtes sautées est dans $\mathcal{O}(|\Sigma|n)$

1. Soit une marche débutant sur une arête (u, v) tel que $f(u)$ est de type a. Soit β le facteur étiquetant (u, v). On sait que le chemin de $f(u)$ à $f(v)$ doit avoir l'étiquette de chemin β. De plus, aucun nœud sur ce chemin ne peu avoir de lien suffixe entrant de type a. En effet, si aw est le facteur associé à l'état u, il faudrait alors que $w\beta'$ avec β' préfixe de β soit une état explicite mais il n'y a pas d'état explicite pour ce mot car l'arête (u, v) passe directement de l'état pour le mot aw à l'état pour le mot $aw\beta$. Donc comme la marche s'arrête quelque part en lisant le facteur β elle ne peut pas sauter une arête terminant par un état avec un lien suffixe entrant de type a.

2. Pour sauter e après avoir suivit un lien suffixe de type a, on doit commencer à sauter et compter après $f(u)$ sinon il faudrait sauter une arête terminant par un état avec un lien suffixe entrant de type a, ici l'état $f(u)$. Par la suite, pour sauter l'arête e, on doit suivre le chemin donné par β, l'étiquette de l'arête (u, v), sinon on ne passera pas par e. Donc la marche doit démarrer sur l'arête (u, v).

3. Le départ d'un tel chemin doit forcément être le premier nœud au-dessus de l'arête e qui a un lien suffixe de type a entrant. Notons u ce nœud. Supposons qu'il y ait deux noeud v_1 et v_2 sous e tel que u vers v_1 et u vers v_2 forme des chemins comme désirés. Soit u' tel que $f(u') = u$ et le lien est de type a. Soit v'_1 et v'_2 définis de façon analogue. On doit donc avoir une nœud v de branchement sous u' et au-dessus v'_1 et v'_2 sinon il y aurait contradiction avec le fait que v_1 et v_2 définisse des chemins tel que voulu (voir Figure 7). Or le lien suffixe de v ne peut être envoyé sur aucun nœud car celui-ci devrait être entre u et v_1. On a donc une contradiction et v_1 et v_2 ne peuvent pas être distinct. Il existe donc au plus un chemin.

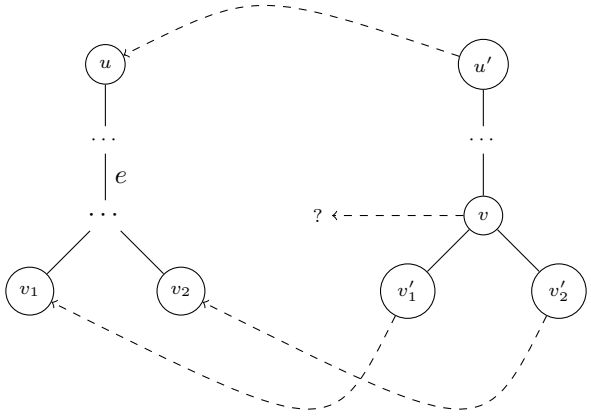


FIGURE 7 – Liens suffixes de type a entre deux portions de l'arbre en pointillé

4. Soit e une arête. Celle-ci est sur au plus un chemin comme définit au point 3. Donc, au plus une arête (u, v) est telle que $f(u)$ et $f(v)$ définissent ce chemin et soit des liens suffixes de type a. On sait par le Lemme 6 qu'il y a au plus deux carrés qui se terminent sur (u, v). Comme chaque marche débute à partir de l'état d'un carrés et que le point 2 indique que chaque marche suivant un lien suffixe de type a et sautant a commence à l'arête (u, v), on a que l'arête e est sautée par au plus deux marches suivant un lien suffixe de type a.

5. Comme il y a $|\Sigma|$ types de liens suffixes possibles, chaque arête est sautée au plus $2|\Sigma|$ fois. Finalement, le nombre d'arêtes est en $\mathcal{O}(n)$ et donc le nombre d'arêtes total sautées est dans $\mathcal{O}(|\Sigma|n)$. □

Théorème 6. *La phase de marquage complet s'effectue en temps et en espace linéaire.*

Démonstration. Le parcours de l'arbre s'effectue en temps linéaire et on effectue une marche suffixe pour chaque type de carré. Comme suivre un lien suffixe s'effectue en temps constant et le nombre de carrés est linéaire, le temps demandé pour suivre tous les liens suffixes est linéaire. On peut sauter une arête en temps constant. Par le Lemme 7, le temps demandé par l'ensemble des procédures sauter et compter est donc linéaire. On a donc que le temps pour effectuer les marches suffixes est linéaire. Finalement, le temps pour décider si on doit démarrer une autre marche est constant car il suffit de vérifier si le carré courant est déjà enregistré sur l'arête (par le Lemme 6, il y en a au plus deux d'enregistrés sur une arête). Cette vérification s'effectue donc en temps constant.

L'espace pour enregistrer le marquage complet est le seul espace supplémentaire occupé par cette phase. Comme le nombre de marques est linéaire, l'espace occupé est aussi linéaire. □

6 Conclusion

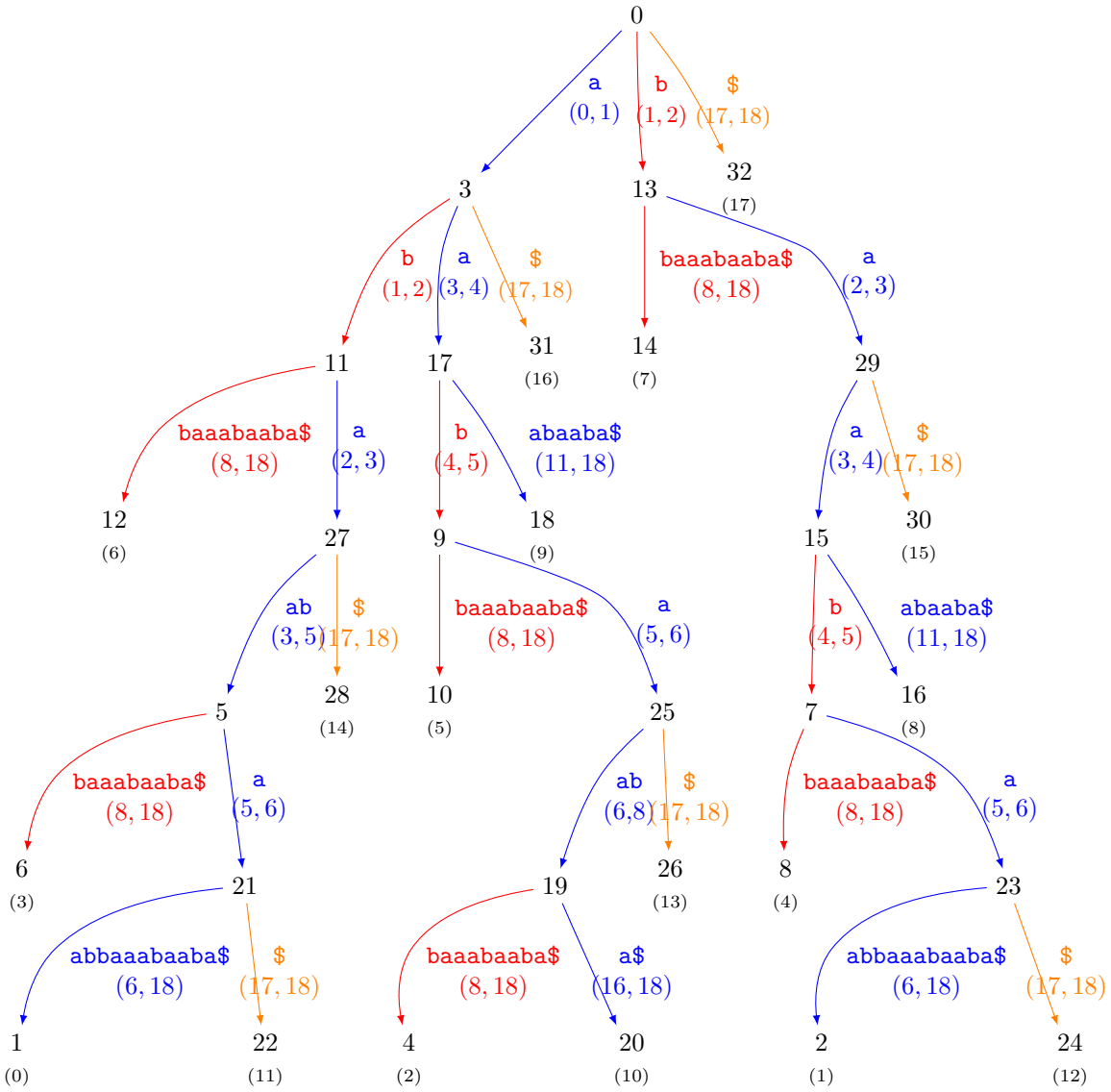
En utilisant la décoration, on peut alors atteindre notre objectif. Un parcours de l'arbre en temps linéaire permet de trouver une paire pour chaque type de carré. En entrelaçant ce parcours avec un parcours du sous-arbre pour chaque type de carré marqué dans l'arbre on peut obtenir toutes les paires de carré de S. Ces parcours de sous-arbres s'effectuent au total en $\mathcal{O}(k)$ du nombre de carrés. On a donc le théorème suivant :

Théorème 7. *On peut obtenir en $\mathcal{O}(n)$ une occurrence de chaque type de carré de S et toute les occurences de carré dans S en $\mathcal{O}(n+k)$ si k dénote le nombre de carré de S.*

En propageant des informations sur les carrés vers les feuilles de l'arbre on peut répondre en temps linéaire à plusieurs questions comme le nombre de carrés commençant à une position ou encore de trouver le plus long ou le plus court carré commençant à une position.

L'implémentation complète de l'algorithme pour le logiciel SAGE est disponible en Annexe B. Le code est en cours d'intégration au logiciel SAGE.

A Arbre suffixe de *abaabaabbaaabaaba*$



B Implémentation

```
#To install tqdm run sage -pip install tqdm in a terminal
from tqdm import tqdm

#============================================================================
#         Fonction à ajouter à Implicit Suffix Tree
#============================================================================
def LZ_decomposition(self):
    r"""
    Return the Lempel-Ziv decomposition of the self.word() in the form of a list
    iB of index such that the blocks of the decomposition are
    self.word()[iB[k]:iB[k+1]]

    The Lempel-Ziv decomposition is the factorisation u_1...u_k of a word
    w=x_1...x_n such that u_i is the longest prefix of u_i...u_k that has an
    occurence starting before u_i or a letter if the prefix is empty.

    EXAMPLE:

        sage: w = Word('abababb')
        sage: T = w.suffix_tree()
        sage: T.LZ_decomposition()
        [0, 1, 2, 6, 7]
        sage: w = Word('abaababacabba')
        sage: T = w.suffix_tree()
        sage: T.LZ_decomposition()
        [0, 1, 2, 3, 6, 8, 9, 11, 13]
        sage: w = Word([0,0,0,1,1,0,1])
        sage: T = w.suffix_tree()
        sage: T.LZ_decomposition()
        [0, 1, 3, 4, 5, 7]
        sage: w=Word('0000100101')
        sage: T=w.suffix_tree()
        sage: T.LZ_decomposition()
        [0, 1, 4, 5, 9, 10]
    """
    iB=[0]
    i=0
    w=self.word()
    while i<len(w):
        l=0
        ((x,y),successor)=self._find_transition(0, w[i])
        x=x-1
        while x<i+l:
            if y==None:
                l=len(w)-i
            else:
                l+=y-x
            if i+l>=len(w):
                l=len(w)-i
                break
            ((x,y),successor)=self._find_transition(successor,w[i+l])
            x=x-1
        i+=max(1,l)
        iB.append(i)
    return iB

def leftmost_covering_set(self):
    r"""
    Compute the leftmost covering set of squares pair in self.word(). Return
    square as pair (i,l) specifying self.word()[i:i+l]

    A leftmost covering set is a set such that the leftmost occurence (j,l) of a
    type of square in self.word() is covered by a pair (i,l) in the set for all
    types of squares. We say that (j,l) is covered by (i,l) if (i,l), (i+1,l),
    ..., (j,l) are all squares.
```

```
The set is return in the form of a list P such that P[i] contains all the
the length of square starting at i in the set. The list P[i] are sort in
decreasing order.

EXAMPLES:

    sage: w=Word('abaabaabbaaabaaba')
    sage: T=w.suffix_tree()
    sage: T.leftmost_covering_set()
    [[6], [6], [2], [], [], [], [], [2], [], [], [6, 2], [], [], [], [], [], []]
    sage: w=Word('abaca')
    sage: T=w.suffix_tree()
    sage: T.leftmost_covering_set()
    [[], [], [], [], []]

REFERENCE:

* [1] Gusfield, D., & Stoye, J. (2004). Linear time algorithms for finding
  and representing all the tandem repeats in a string. Journal of Computer
  and System Sciences, 69(4), 525-546.
"""
def condition1_square_pairs(i):
    r"""
    Compute the square that has their center in the i-th block of
    LZ-decomposition and that start in the i-th block and end in the
    (i+1)-th
    """
    for k in range(1,B[i+1]-B[i]+1):
        q=B[i+1]-k
        k1=w.longest_forward_extension(B[i+1],q)
        k2=w.longest_backward_extension(B[i+1]-1,q-1)
        start=max(q-k2,q-k+1)
        if k1+k2>=k and k1>0 and start>=B[i]:
            #print "Condition 1 yield (%s,%s) for block %s" %(start,2*k,i)
            #print start>=B[i]
            yield (start,2*k)

def condition2_square_pairs(i):
    r"""
    Compute the squares that has their center in the i-th block of the
    LZ-decomposition and that starts in the (i-1)-th block or before. Their
    end is either in the i-th or the (i+1)-th block
    """
    try:
        end=B[i+2]-B[i]+1
    except IndexError:
        end=B[i+1]-B[i]+1
    for k in range(2,end):
        q=B[i]+k
        k1=w.longest_forward_extension(B[i],q)
        k2=w.longest_backward_extension(B[i]-1,q-1)
        start=max(B[i]-k2,B[i]-k+1)
        #print "k=%s q=%s k1=%s k2=%s, start=%s, h1=%s" %(k,q,k1,k2,start,B[i+1])
        if k1+k2>=k and k1>0 and start+k<=B[i+1] and k2>0:# and start<B[i]:
            #print "Condition 2 yield (%s,%s) for block %s" %(start,2*k,i)
            #print start<B[i]
            yield (start,2*k)

w=self.word()
B=self.LZ_decomposition()
P=[[] for _ in w]
for i in range(len(B)-1):
    squares=list(condition2_square_pairs(i))+list(condition1_square_pairs(i))
    for (i,l) in squares:
        P[i].append(l)
for l in P:
    l.reverse()
return P
```

```
def count_and_skip(self,node,(i,j)):
    r"""
    Use count and skip trick to follow the path starting and "node" and
    reading self.word()[i:j]. We assume that reading self.word()[i:j] is
    possible from "node"

    INPUTS:

        node - explicit node of T
        (i,j) - Indices of factor T.word()[i:j]

    OUTPUT:

        The node obtained by starting at "node" and following the edges
        labeled by the letter of T.word()[i:j]. Returns "("explicit",
        end_node) if w ends at a "end_node", and "("implicit", edge, d)" if
        it ends at a spot along an edge.

    EXAMPLES:

        sage:T=Word('00110111011').suffix_tree()
        sage:T.count_and_skip(5,(2,5))
        ("implicit", (9, 10), 2)
        sage:T.count_and_skip(0, (1, 4))
        ("explicit", 7)
    """
    if i==j: #We're done reading the factor
        return ('explicit',node)
    transition=self._find_transition(node,self._letters[i])
    child=transition[1]
    if transition[0][1]==None: #The child is a leaf
        edge_length=len(self.word())-transition[0][0]+1
    else:
        edge_length=transition[0][1]-transition[0][0]+1
    if edge_length>j-i: #The reading stop on this edge
        return ('implicit',(node,child),j-i)
    return self.count_and_skip(child,(i+edge_length,j))

def suffix_walk(self,(edge,l)):
    r"""
    Compute the suffix walk from the input state. If the input state is path
    label "aw" with "a" a letter, the output is the state of "w".

    INPUTS:

        edge - the edge containign the state
        l - the string-depth of the state on edge (l>0)

    OUTPUT:

        Returns "("explicit", end_node) if the state of "w" is an explicit state
        and "("implicit", edge, d)" if the state of "w" is implicit on "edge".

    EXAMPLES:

        sage:T=Word('00110111011').suffix_tree()
        sage:T.suffix_walk(((0,5),1))
        ("explicit", 0)
        sage:T.suffix_walk(((7,3),1))
        ("implicit", (9,4), 1)
    """
    #If the state is implicit
    parent=self.suffix_link(edge[0])
    for (i,j) in self._transition_function[edge[0]]:
        if self._transition_function[edge[0]][(i,j)]==edge[1]:
            break
    #(i-1,j) is the label of edge
```

```
    i-=1
    return self.count_and_skip(parent,(i,i+l))


from sage.combinat.words.suffix_trees import ImplicitSuffixTree
ImplicitSuffixTree.LZ_decomposition = LZ_decomposition
ImplicitSuffixTree.leftmost_covering_set = leftmost_covering_set
ImplicitSuffixTree.count_and_skip=count_and_skip
ImplicitSuffixTree.suffix_walk=suffix_walk

#============================================================================
#        Fonction à ajouter Word
#============================================================================
def longest_forward_extension(self,x,y):
    r"""
    Compute the length of le longest factor of self that starts at x and that
    matches a factor that starts at y. Returns 0 if x or y are not valid
    position in self.

    INPUTS:

        x,y - positions in self

    EXAMPLES:

        sage:w=Word('0011001')
        sage:w.longest_forward_extension(0,5)
        3
        sage:w.longest_forward_extension(0,2)
        0
        sage:w.longest_forward_extension(-3,2)
        0
    """
    length=self.length()
    if not (0<=x and 0<=y):
        return 0
    l=0
    while x<length and y<length and self[x]==self[y]:
        l+=1
        x+=1
        y+=1
    return l


def longest_backward_extension(self,x,y):
    r"""
    Compute the length of le longest factor of w that ends at x and that
    matches a factor that ends at y. Returns 0 if x or y are note valid position
    in self.

    INPUTS:

        x,y - positions in self

    EXAMPLES:

        sage:w=Word('0011001')
        sage:w.longest_backward_extension(7,2)
        3
        sage:w.longest_backward_extension(1,5)
        1
        sage:w.longest_forward_extension(4,23)
    """
    length=self.length()
    if not (x<length and y<length):
        return 0
    l=0
    while x>=0 and y>=0 and self[x]==self[y]:
```

```
        l+=1
        x-=1
        y-=1
    return l

from sage.combinat.words.finite_word import FiniteWord_class
FiniteWord_class.longest_forward_extension=longest_forward_extension
FiniteWord_class.longest_backward_extension=longest_backward_extension

#============================================================================
#         Nouvelle classe
#============================================================================

class DecoratedSuffixTree(ImplicitSuffixTree):

    def __init__(self, w):
        r"""
        Construct the decorated suffix tree of a word

        A decorated suffix tree of w is the suffix tree of w  marked with the
        end point of all squares in the w.

        The symbol "$" is append to w to ensure de that each final state is a
        leaf of the suffix tree.

        When using pair as output, all the algorithm are linear in the length of
        the word w

        INPUT:

            w - word

        EXAMPLES:

            sage: w=Word('0011001')
            sage: DecoratedSuffixTree(w)
            Decorated suffix tree of : 0011001$

        REFERENCE:

          * [1] Gusfield, D., & Stoye, J. (2004). Linear time algorithms for
          finding and representing all the tandem repeats in a string. Journal
          of Computer and System Sciences, 69(4), 525-546.
        """
        if not isinstance(w, FiniteWord_class):
            raise ValueError("w must be a member of FiniteWord_class")
        if "$" in w:
            raise ValueError("The symbol '$' is reserved for this class ")
        end_symbol="$"
        w = Word(str(w)+"$")
        ImplicitSuffixTree.__init__(self, w)
        self.labeling=self._complete_labeling()

    def __repr__(self):
        w=self.word()
        if len(w)>40:
            w=str(w[:40])+'...'
        return "Decorated suffix tree of : %s" %w

    def _partial_labeling(self):
        r"""
        Make a depth first search in the suffix tree and mark some squares of a
        leftmost covering set of the tree. Used by _complete_labeling.

        EXAMPLES:

            sage:w=Word('abaababbabba')
            sage:T=DecoratedSuffixTree(w)
```

```
    sage:T._partial_labeling()
    {(3, 4): [1], (5, 1): [3], (5, 6): [1], (11, 17): [1], (13, 8): [1], (15, 10):
        [2]}
"""
def node_processing(node,parent,(i,pos)):
    r"""
    Mark point along the edge (parent,node) if the string depht of parent is
    smaller than the lenght of the tamdem repeat at the head of P(node).
    Make it for all such squares pairs and remove them from P(node).
    P(node)=P[i][pos:]
    INPUTS:
        node - a node of T
        parent - the parent of node in T
        (i,pos) - the pair that represent the head of the list P(node)
    OUTPUT:
        (i,pos) - the new head of P(node)
    """
    while pos<len(P[i]) and P[i][pos]>string_depth[parent]:
        label=P[i][pos]-string_depth[parent]
        try:
            labeling[(parent,node)].append(label)
        except KeyError:
            labeling[(parent,node)]=[label]
        pos+=1
    return (i,pos)

def treat_node(current_node,parent):
    r"""
    Proceed to a depth first search in T, couting the string_depth of
    each node a processing each node for marking

    To initiate de depth first search call treat_node(0,None)

    INPUTS:

        current_node - A node
        parent - Parent of current_node in T

    OUTPUT:

        The resulting list P(current_node) avec current_node have been
        process by node_processing. The ouput is a pair (i,pos) such
        that P[i][pos:] is the list of current_node.
    """
    #Call recursively on children of current_node
    if D.has_key(current_node):
        node_list=(n,0)
        for child in D[current_node].iterkeys():
            (i,j)=D[current_node][child]
            if j==None:
                j=n
            string_depth[child]=string_depth[current_node]+j-i
            child_list=treat_node(child,current_node)
            if child_list[0]<node_list[0]:
                node_list=child_list
    else: #The node is a child
        node_list=(n-string_depth[current_node],0)
    #Make teatement on current node hear
    return node_processing(current_node,parent,node_list)

P=leftmost_covering_set(self)
D=self.transition_function_dictionary()
string_depth=dict([(0,0)])
n=len(self.word())
labeling=dict()
treat_node(0,None)
return labeling
```

```
def _complete_labeling(self):
    r"""
    Returns a dictionnary of edges of self, with markpoint for the end of
    each square types of T.word()

    INPUT:

        self - Suffix tree

    EXAMPLES:
        sage:w=Word('aabbaaba')
        sage:DecoratedSuffixTree(w)._complete_labeling()
        {(2, 7): [1], (5,4): [1]}
    """

    def walk_chain(u,v,l,start):
        r"""
        Execute a chain of suffix walk until a walk is unsuccesful or it got
        to a point already register in QP. Register all visited point in Q.

        INPUTS:

            (u,v) - edge on wich the point is registered
            l - depth of the registered point on (u,v)
            start - start of the squares registered by the label (u,v),l
        """
        #Mark the point in labeling
        try:
            labeling[(u,v)].append(l)
        except KeyError:
            labeling[(u,v)]=[l]
        #Make the walk
        final_state=self.suffix_walk(((u,v),l))
        successful=False
        if final_state[0]=='explicit':
            parent=final_state[1]
            transition=self._find_transition(parent,self._letters[start])
            if transition!=None:
                child=transition[1]
                successful=True
                depth=1
        else:
            parent=final_state[1][0]
            child=final_state[1][1]
            depth=final_state[2]
            next_letter=self._letters[D[parent][child][0]+depth]
            if next_letter==self._letters[start]:
                successful=True
                depth+=1
        #If needed start a new walk
        if successful:
            try:
                if depth not in prelabeling[(parent,child)]:
                    walk_chain(parent,child,depth,start+1)
            except KeyError:
                walk_chain(parent,child,depth,start+1)

    def treat_node(current_node,(i,j)):
        r"""
        Execute a depht first search on self and start a suffix walk for
        labeled points on each edges of T. The fonction is reccursive, call
        treat_node(0,(0,0)) to initiate the search

        INPUTS:

            current_node - The node that is to treat
            (i,j) - Pair of index such that the path from 0 to current_node
            reads T.word()[i:j]
```

```
            """
            if D.has_key(current_node):
                for child in D[current_node].iterkeys():
                    edge=(current_node,child)
                    edge_label=D[edge[0]][edge[1]]
                    treat_node(child,(edge_label[0]-(j-i),edge_label[1]))
                    if prelabeling.has_key((current_node,child)):
                        for l in prelabeling[edge]:
                            square_start=edge_label[0]-(j-i)
                            walk_chain(current_node,child,l,square_start)

        prelabeling=self._partial_labeling()
        labeling=dict()
        D=self.transition_function_dictionary()
        treat_node(0,(0,0))
        return labeling

    def square_vocabulary(self,output="pair"):
        r"""
        Return the list of squares in the squares vocabulary of self.word.
        Return a list of pair in output="pair" and the explicit word if
        output="word"

        INPUTS:

            output - "pair" or "word"

        EXAMPLES:

            sage: w=Word('aabb')
            sage: DecoratedSuffixTree(w).square_vocabulary()
            [(0, 0), (0, 2), (2, 2)]
            sage: w=Word('00110011010')
            sage: DecoratedSuffixTree(w).square_vocabulary(output="word")
            [word: , word: 01100110, word: 00110011, word: 00, word: 11, word: 1010]
        """
        def treat_node(current_node,(i,j)):
            if D.has_key(current_node):
                for child in D[current_node].iterkeys():
                    edge=(current_node,child)
                    edge_label=(D[edge[0]][edge[1]])
                    treat_node(child,(edge_label[0]-(j-i),edge_label[1]))
                    if Q.has_key((current_node,child)):
                        for l in Q[(current_node,child)]:
                            square_start=edge_label[0]-(j-i)
                            pair=(square_start,edge_label[0]+l-square_start)
                            squares.append(pair)

        if not(output=="pair" or output=="word"):
            raise ValueError("output should be 'pair' or 'word'; got %s" %output)
        D=self.transition_function_dictionary()
        Q=self.labeling
        squares=[(0,0)]
        treat_node(0,(0,0))
        if output=="pair":
            return squares
        else:
            return [self.word()[i:i+l] for (i,l) in squares]

#============================================================================
#         Pas encore triée
#============================================================================

def naive_square_voc(T):
    r"""Compute the square vocabulary of T.word()
    INPUTS:
        T - Suffix Tree
    OUTPUT:
```

```
        Square vocabulary of T.word()"""
    squares=[]
    for f in [v for v in T.factor_iterator() if v.is_square()]:
        squares.append(f)
    return set(squares)

def run_test(n,alphabet='01',test_for_double=False):
    r"""
    Compare the algorithme with a naive algorithme
    INPUTS:
        n - size of words to test on
        test_for_double - If true detect a bug if the algorithm return a double
        alphabet - the alphabet to test on
    OUTPUT:
        True if works for all words, False if it bugs for a word
    """
    for w in tqdm(Words(alphabet,n)):
        S1=naive_square_voc(w)
        T=DecoratedSuffixTree(w)
        L=T.square_vocabulary(output="word")
        S2=set(L)
        if test_for_double and len(S2)!=len(L):
            print "Problème de doublon avec %s" %w
            print "is_sort_leftmost(%s)=%s" %(w,is_sort_leftmost(w))
            return False
        if not(S2.issubset(S1) and S1.issubset(S2)):
            print "Problème avec %s" %w
            return False
    return True

def is_sort_leftmost(w,strictly=True):
    r"""
    Verify if the leftmost covering set is sort according to the algorithm.
    If strictly is true, the list must be stricly increasing in order
    INPUTS:
        w - A word ending with $
    """
    covering=leftmost_covering_set(w.suffix_tree())
    for l in covering:
        for i in range(len(l)-1):
            if strictly and l[i][1]<=l[i+1][1]:
                return False
            elif l[i][1]<=l[i+1][1]:
                return False
    return True

def LZ_decomposition_explicit(T):
    r"""Take the explicit suffix tree of a word and return the Lempel-Ziv
    decomposition of the word in the form of a list iB of index such that the
    blocks of the decomposition are T.word()[iB[k]:iB[k+1]]"""
    iB=[0]
    i=0
    w=T.word()
    while i<len(T.word()):
        l=0
        s=0
        ((x,y),successor)=T._find_transition(s, w[i])
        x=x-1 #Pourquoi find_transtion retourne pas la bonne étiquette?
        while x<i+l and y!=None and i+l<len(w):
            l+=y-x
            s=successor
            if i+l==len(w):
                break
            transition=T._find_transition(s,w[i+l])
            ((x,y),successor)=transition
            x=x-1 #ici
        i+=max(1,l)
        iB.append(i)
```

```
return iB
```

Références

[1] Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain ? *J. Combin. Theory Ser. A*, 82(1) :112–120, 1998.

[2] Dan Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge University Press, Cambridge, 1997. Computer science and computational biology.

[3] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. System Sci.*, 69(4) :525–546, 2004.

[4] Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. Discrete Algorithms*, 8(4) :418–428, 2010.

[5] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, 1995.