# Leçon 907 : Algorithmique du texte. Exemples et applications.

Julie Parreaux

2018 - 2019

Références pour la leçon

[2] Carton, *Langages formels, calculabilité et complexité.*
[3] Cormen, *Algorithmique.*
[4] Crochemore, Hancart et Lecroq, *Algorithme du texte.*
[5] Crochemore et Rytter, *Text Algorithms.*

Développements de la leçon

L'automate minimal de la recherche de motif — Le calcul de l'alignement optimal

## Plan de la leçon



## Motivation

### Défense

Le texte est omniprésent en informatique. Des fichiers textes permette d'écrire des programmes informatique : on a dû développer des algorithmes afin de concevoir des logiciel ca-

pable de gerer du texte, mais égalent des logiciel capable de compiler nos programme qui sont sous la forme d'un texte. L'algorithmique du texte est ensuite apparue dans la bio-informatique où on a cherché à traiter les chaînes de l'ADN ou lors de traitement multimédia lorsque l'on recherche à compresser toutes ces données ou encore en sécurité avec la protection des mots de passes.

On a une algorithmique complexe et très variées qui touche à toutes sorte de problèmes.

### Ce qu'en dit le jury

Cette leçon devrait permettre au candidat de présenter une grande variété d'algorithmes et de paradigmes de programmation, et ne devrait pas se limiter au seul problème de la recherche d'un motif dans un texte, surtout si le candidat ne sait présenter que la méthode naïve.

De même, des structures de données plus riches que les tableaux de caractères peuvent montrer leur utilité dans certains algorithmes, qu'il s'agisse d'automates ou d'arbres par exemple.

Cependant, cette leçon ne doit pas être confondue avec la 909, « Langages rationnels et Automates finis. Exemples et applications. ». La compression de texte peut faire partie de cette leçon si les algorithmes présentés contiennent effectivement des opérations comme les comparaisons de chaînes : la compression LZW, par exemple, est plus pertinente dans cette leçon que la compression de Huffman.

## Introduction

— *Motivation* : Compilation, traitement de texte, stockage des données

— *Définition - Notation* : alphabet + mot + préfixe

## 1 Recherche de motif

Lorsqu'on a un texte, on souhaite pouvoir retrouver rapidement un mot (ou un motif) dans celui-ci.

— *Notations* : texte $T$ et motifs $P$

— *Remarque préliminaire* : si $P$ est un motif dans $T$ alors $|P| \leq |T|$.

— *Applications* : fouille de données ; contrôle F sur un ordinateur ; ...

### 1.1 Recherche d'un motif exact

On commence par recherche le motif exacte (on suppose qu'il n'y a pas d'erreur de frappe, d'orthographe dans le motif ou dans le texte). On va étudier toutes sorte d'algorithme étant plus ou moins efficace pour répondre à cette question.

#### L'algorithme naïf [3, p.905]

— *Principe* : on teste tous les caractère jusqu'à ce qu'on trouve (fenêtre glissante)

— Algorithme 1 + Complexité en temps et en espace

— On peut et on veut faire mieux

| Algorithme | Prétraitement | Recherche | Espace |
|---|---|---|---|
| Naïf | 0 | $O(tp)$ | $O(1)$ |
| Rabin–Karp | $\Theta(p)$ | $O(tp)$ | $O(1)$ |
| Boyer-Moore | 0 | $O(tp)$ | $O(1)$ |
| Automate | $O(m\|\Sigma\|)$ | $O(t)$ | $O(m\|\Sigma\|)$ |
| Morris–Pratt | $O(p\|\Sigma\|)$ | $\Theta(t)$ | $O(p)$ |
| Knuth–Moris–Pratt | $\Theta(p)$ | $\Theta(t)$ | $O(p)$ |
| Arbre | $O(t)$ | $O(m)$ | $O(n \log n)$ |

TABLE 1 – Récapitulatif des complexités pour les algorithmes de recherche de motifs.

**L'algorithme Boyer–Moore** Cet algorithme est intéressant car il ressemble beaucoup à l'algorithme naïf mais au lieu de lire le motif de gauche à droite, il le lit de droite à gauche. En effet, si les dernière lettre ne correspondent pas, il ne peut pas être à cette place. Ensuite, on le décale du nombre de lettre lues.

— *Principe* : on lit le motif dans l'autre sens

— Complexité en temps et en espace

**Utilisation d'un automate des occurrences [3, p.915], [4, p.182]**

— *Principe* : Trouver $P$ dans $T$ c'est trouver tous les $T_i$ ayant $P$ pour suffixe : $\{i : T_i \in \Sigma^* P\}$. On construit donc un automate des occurrences (exemple)

— *Notations* : $\phi(T_i)$ l'état après la lecture de $T_i$ et $\sigma(T_i)$ le plus long suffixe de $P$ qui est préfixe de $T_i$

— *Théorème* : $\sigma(T_i) = \phi(T_i)$ L'automate des occurrences est minimal pour reconnaître $\Sigma^* P$. DEV

— *Extension* : reconnaître plusieurs mots (automate en parallèle)

**Utilisation de la notion de bords**

— *Définition* : bord + exemple [4, p.11]

— *Principe* : on peut calculer d'où recommencer la recherche avec le bord du motif (on fait moins de comparaison, on est donc plus rapide)

— Algorithmes : MP (Algorithme 6) et KMP (Algorithme 7)(qui améliore ce procédé) + complexités (inconvénient de la méthode)

**Utilisation d'un arbre à suffixe [4, p.162]**

## 1.2 Cas d'une expression régulière (motif incomplet)

— *Problème* : $P$ peut être considéré comme une expression régulière

— *Exemple* *.ml dans une liste de fichiers

— *Principe* : Construire l'automate fini qui la reconnaît, le déterminiser et l'appliquer.

— *Complexité* : Exponentielle dans le pire cas (en pratique cela marche raisonnablement)

— *Application* : grep

— *Application* : Analyse lexicale à l'aide de l'automate des motifs constitué d'expressions régulières

## 2 Comparaison de textes

*Applications* : correcteurs orthographiques, bio-informatique (étude du génome), traitement du signal, analyse de fichier (détection de virus), ...

### 2.1 Plus longue sous-séquence commune [3, p.362]

— Spécification du problème et exemple

— Algorithme et complexités

### 2.2 Distance d'édition et recherche approchée [4, p.224]

La distance d'édition permet de comparer deux textes à priori distincts et de classer à quels points les deux textes sont distincts. Le calcul de telles distances peut-être délicat. On utilise alors des paradigmes issues de la programmation dynamique. Outre l'utilisation en bio-informatique (via les études sur l'ADN), les distances d'éditions sont également utilisées dans la recherche approchées de motifs.

— *Définitions* : fonctions de substitution, d'ajout et de suppression ; distance d'édition de Levenstein

— *Théorème* : caractérisation de la notion de distance

— *Définition* : Alignement + exemple + *Définition* : coût d'un alignement

— *Algorithme* : Calcul de l'alignement optimal entre deux mots DEV

— *Application* : problème de la recherche à $k$ différences.

### 2.3 Répétition dans un mot [4, p.211]

— *Problème* : Recherche de facteurs se répétant dans un mots

— *Méthode* : automate des occurrences et recherche d'un état particulier.

## 3 Compression [5]

Lorsqu'on stocke un grand volume de données, on souhaite les compresser afin d'en réduire la place. La compression du texte est la première compression que nous avons réalisée : tout fichier informatique peut être vu comme du texte. Aujourd'hui chaque compression est spécifique au type des données que l'on stocke.

— *Remarque* : on ne peut pas compresser indéfiniment sans perte (théorème de Shannon en théorie de l'information)

— Algorithme LZ + exemple + remarque sur son utilité actuelle

— Algorithme de LZW

LZH + perte ?

## 4 Traitement d'un texte avec une grammaire

Les informaticiens écrivent beaucoup de textes pour discuter avec une machine. Les langages de programmation sont les premiers à nécessité une transformation via un algorithme de texte.

— Processus de compilation : analyse syntaxique

— *Définition* : problème du mot + forme normale

— *Théorème* : Décidabilité du problème du mot avec une grammaire algébrique sous forme FNC + algorithme de CYK.

## Ouverture

La compression fait appelle à de nombreux autres formats : ZIP ; ΔDeleta (git) ; ...
Le problème de l'encodage : UTF8 ; ASCII
Quelques problèmes non-abordé : Recherche de motif incomplet, ....

## Quelques notions importantes

### La recherche de motifs exacte

*Objectif* : Recherche d'un élément lorsque $K$ est l'ensemble des mots de $|\Sigma|^*$ (où $\Sigma$ est un alphabet) et $E$ est l'ensemble des facteurs d'un texte.

*Structure de données associée* : tableau de longueur la taille de la chaîne de caractère considérée. On notera $T = [1 \dots t]$ le tableau du texte et $P = [1 \dots p]$ le motif rechercher dans $T$ où $t$ est la taille du texte et $p$ est celle du motif.

*Remarques préliminaires* : Le motif ne peut être contenue dans le texte uniquement si sa taille est inférieure à celle du texte et s'ils sont définis sur un alphabet commun que l'on note $\Sigma$. Ce que l'on supposera dans la suite.

**Algorithme naïf [3, p.905]** Cet algorithme est loin d'être optimal en complexité temporelle. Par contre, il a une très bonne complexité spatiale et ne nécessite aucun pré-traitement.

**Algorithm 1** Recherche naïve d'un motif dans un texte.

```
1:  function RECHERCHE-NAIVE(T, P) ▷ T le texte et P le motif
2:      t ← T.longueur
3:      p ← P.longueur
4:      for s = 0 à t − p do
5:          if P[1, ..., m] = T[s + 1, ..., s + m] then
6:              Retourner s
7:          end if
8:      end for
9:      Retourner 0
10: end function
```

*Principe* : L'algorithme naïf utilise le principe de la fenêtre glissante. Il va chercher le motif en décalant de un caractère en cas d'erreur. De plus la comparaison motif-texte se fait de gauche à droite. Il renvoie le décalage représentant le début du motif dans le texte ou 0 si le motif n'apparaît pas dans le texte.
*Complexité dans le pire cas* : $O((t-p+1)p)$ ou l'ensemble des positions possibles pour le motif et la taille du motif que nous devons tester à chaque fois.
On peut attendre la borne atteinte de cet algorithme avec $T = a^{p-1}ba^p$ et $P = a^p$.
*Complexité spatiale* : $O(1)$.

**Algorithme de Rabin-Karp [3, p.905]** Algorithme qui se comporte bien en moyenne pour la recherche de chaîne. De plus, il se généralise bien à des problèmes voisins tels que la recherche des motifs bidimensionnels.

*Hypothèse 1* : Le motif est fixe et connu à l'avance.

*Hypothèse 2* : On considère que chacun des caractères est exprimé comme un chiffre en base $d = |\Sigma|$. Dans le cadre de cette leçon, on considère que $\Sigma = \{0, 1, \ldots, 9\}$. (Cette hypothèse n'est pas très contraignante, il suffit de travailler en base $|\Sigma|$ puisque l'alphabet est fini)

**Algorithm 2** Algorithme de Rabin-Karp pour la recherche d'un motif dans un texte.

```
1:  function RABIN-KARP(T, P, d, q)        ▷ T le texte, P le motif, d = |Σ|, q un nombre premier
2:      t ← T.longueur ; p ← P.longueur
3:      h ← d^{p+1} mod q
4:      p ← 0 ; t_0 ← 0
5:      for i = 1 à p do                    ▷ Pré-traitement
6:          p ← (dp + P[i]) mod q           ▷ Valeur du motif
7:          t_0 ← (dt_0 + T[i]) mod q       ▷ Valeur du 1er bloc
8:      end for
9:      for s = 0 à t − p do                ▷ Recherche de la correspondance
10:         if p = t_s then
11:             if P[1, ..., p] = T[s + 1, ..., s + p] then
12:                 Retourner s
13:             end if
14:             t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
15:         end if
16:     end for
17:     Retourner 0
18: end function
```

*Principe* : Cet algorithme s'appuie sur la théorie élémentaire des nombres. En effet, tout bloc de la taille du motif dans le texte ainsi que le motif vont être soumis à un calcul donnant une valeur modulo un nombre premier. A la place de tester si tous les blocs sont égaux au motif, on testera si tous les blocs qui ont le même nombre modulo ce nombre premier sont égaux. On espère avoir moins de tests à effectuer. L'algorithme renvoie le décalage représentant le début du motif dans le texte ou 0 si le motif n'apparaît pas dans le texte.

*Remarque* : On peut appliquer le même principe avec une fonction de hachage [5, p.367]. Dans ce cas, l'hypothèse 2 n'est plus nécessaire.

*Complexité du prétraitement* : $\Theta(p)$ Les valeurs pour le texte sont calculées à la volée lors de la recherche. Le prétraitement permet de calculer la valeur du motif et la première valeur pour le texte.

*Complexité dans le pire cas* : $O((t - p + 1)p)$ Comme dans le cas naïf. On peut atteindre la borne avec $T = 002002011$ et $P = 011$.

*Complexité spatiale* : $O(1)$ (On stocke la valeur du motif et la valeur du bloc précédent)

*Application* : Le problème de recherche multiples motifs se résout avec un lourd prétraitement sur le texte $T$ ou les multiples motifs $P_i$ afin de comparer les multiples $P_i$ à une portion de texte une seule fois. On adapte l'algorithme de Rabin-Karp pour résoudre efficacement ce problème. (Autre algorithme pour ce problème : l'algorithme d'Aho-Corasick.)

*Utilisation* : détection de plagiat ; comparaison d'un fichier suspect à des fragments de virus ; ...

**Algorithme de Knuth-Morris-Pratt [3, p.905]** Les algorithmes Morris–Pratt et Knuth–Morris–Pratt sont des algorithmes qui utilisent la notion de bord. On utilise le bord des préfixes du motif pour être plus astucieux et rapide quand une comparaison échoue : il nous donne le décalage que l'on doit réaliser.

*Hypothèse* : Le motif est fixe et connu à l'avance.

**Quelques notions sur le bord [1, p.340]** Le bord a un rôle essentiel dans ces algorithmes : il permet de calculer le décalage que l'on va effectuer lors d'un échec de comparaison. Bien définir cette notion est donc primordiale.

**Définition.** Un bord de $x$ est un mot distinct de $x$ qui est à la fois préfixe et suffixe de $x$. On note $Bord(x)$ le bord maximal d'un mot non vide.

*Exemple* Le mot *ababa* possède les trois bords $\epsilon, a$ et $aba$. De plus $Bord(ababa) = aba$.

**Proposition.** *Soit $x$ un mot non vide et $k \in \mathbb{N}$, le plus petit, tel que $Bord^k(x) = \epsilon$.*

1. *Les bords de $x$ sont les mots $Bord^i(x)$ pour tout $i \in \{1, \dots, k\}$.*
2. *Soit $a$ une lettre. Alors, $Bord(xa)$ est le plus long préfixe de $x$ qui est dans l'ensemble $\{Bord(x)a, \dots Bord(x)^k a, \epsilon\}$.*

*Arguments de la preuve.* 1. $z$ est un bord de $Bord(x)$ ssi $z$ est un bord de $x$ (+ récurrence).
2. $z$ est un bord de $za$ ssi $z = \epsilon$ ou $z = z'a$ avec $z'$ un bord de $x$.

□

**Corollaire.** *Soit $x$ un mot non vide et soit $a$ une lettre. Alors,*

$$Bord(xa) = \begin{cases} Bord(x)a & \text{si } Bord(x)a \text{ est préfixe de } x \\ Bord(Bord(x)a) & \text{sinon} \end{cases}$$

*Arguments de la preuve.* — Si $Bord(x)a$ est préfixe de $x$ alors $Bord(x)a = Bord(xa)$.
— Sinon, $Bord(xa)$ est préfixe de $Bord(x) = y$ et le plus long préfixe de $x$ dans $\{Bord(y)a, \dots Bord(y)^k a, \epsilon\}$.

□

**Définition.** Soit $x$ un mot de longueur $m$. On pose $\beta : \{0, \dots, m\} \to \{-1, \dots, m-1\}$ la fonction qui est définie comme suit : $\beta(0) = -1$ et pour tout $i > 0$, $\beta(i) = |Bord(x_1, \dots, x_i)|$. On pose $s : \{1, \dots, m-1\} \to \{0, \dots, m\}$ la fonction suppléance de $x$ définie comme suit : $s(i) = 1 + \beta(i-1), \forall i \in \{1, \dots, m\}$.

*Remarque* : on a bien entendu $\beta(i) \leq i - 1$.

**Corollaire.** *oit $x$ un mot non vide de longueur $m$. Pour $j \in \{0, \dots, m-1\}$, on a $\beta(1+j) = 1 + \beta^k(j)$ où $k \geq 1$ est le plus petit entier vérifiant l'une des deux conditions suivantes :*

1. $1 + \beta^k(j) = 0$
2. $1 + \beta^k(j) \neq 0$ *et* $x_{1+\beta^k(j)} = x_{j+1}$

*Arguments de la preuve.* Algorithme 3 □

**Algorithm 3** Calcul de la fonction $\beta$ donnant la taille des bords maximaux d'un mot $x$.

```
1:  function BORD-MAXIMAUX(x, β)        ▷ x mot de taille m
2:      β[0] ← −1
3:      for j = 1 à m do
4:          i ← β[j − 1]
5:          while i ≥ 0 et x[j] ≠ x[i + 1] do
6:              i ← β[i]
7:          end while
8:          β[j] ← i + 1
9:      end for
10: end function
```

**Algorithm 4** Calcul de la fonction suppléance $s$ du mot $x$.

```
1:  function SUPPLÉANCE(x, s)        ▷ x mot de taille m
2:      s[1] ← 0
3:      for j = 1 à m − 1 do
4:          i ← s[j]
5:          while i ≥ 0 et x[j] ≠ x[i] do
6:              i ← s[i]
7:          end while
8:          s[j + 1] ← i + 1
9:      end for
10: end function
```

**L'algorithme de Morris–Pratt [3, p.916]** L'algorithme de Morris–Pratt utilise un automate des occurrences pour calculer les bords du motif. On définit l'automate des occurrences associé à $P[1 \dots m]$ comme suit : $\mathcal{Q} = \{1, \dots m\}$ ; $q_0 = 0$ ; $F = \{m\}$ ; $\delta(q, a) = \sigma(P_q a)$ pour tout $a \in \Sigma$ et $q \in \mathcal{Q}$.

**Définition.** La fonction suffixe associé au motif $P$ $\sigma : \Sigma^* \mapsto \{0, 1, \dots, m\}$ donne la longueur de $Bord(x)$ pour $x \in \Sigma^*$.

**Algorithm 5** Calcul de la fonction de transition $\delta$ de l'automate des occurrences.

```
1:  function CALCUL-δ(P, Σ)   ▷ P motif ; Σ alphabet
2:      p ← P.longueur
3:      for q = 0 à p do
4:          for a ∈ Σ do
5:              k ← min(p + 1, q + 2)
6:              repeat
7:                  k ← k + 1
8:              until P_k préfixe de P_q a   ▷ P_q est le préfixe de P de longueur q.
9:              δ(q, a) ← k
10:         end for
11:     end for
12:     Retourner δ
13: end function
```

*Complexité temporelle du prétraitement dans le pire cas* : $O(p|\Sigma|)$ Ce fait une fois par motif.

**Algorithm 6** Calcul de la recherche dans l'automate des occurrences.

```
1:  function MORRIS–PRATT(T, δ, p)   ▷ T texte ; δ fonction transition ; p la longueur du motif
2:      t ← T.longueur
3:      for i = 1 à t do
4:          q ← δ(q, T[i])
5:          if q = p then
6:              Retourner i − m
7:          end if
8:      end for
9:      Retourner 0
10: end function
```

*Complexité temporelle dans le pire cas* : $\Theta(t)$

*Remarque* : La validité de cet algorithme provient de la validité de la construction de l'automate des occurrences.

**L'algorithme de Knuth–Morris–Pratt [1, p.343]** Cet algorithme utilise une méthode plus astucieuse afin de calculer les bords du préfixes. On s'épargne ainsi un calcul de l'automate des occurrences et on calcul la fonction $\delta$ puisse à la volée. On exploite les propriétés de la fonction de bord.

**Définition.** La fonction préfixe associé au motif $P$ $\pi : \{1, 2, \dots, p\} \mapsto \{0, 1, \dots, p-1\}$ donne la longueur du plus long préfixe de $P$ qui est suffixe propre de $P_q$ où $q \in \{1, 2, \dots, p\}$.

Si on ne fait pas l'hypothèse d'un préfixe propre alors le plus long suffixe d'un mot qui est aussi son préfixe est le mot en question. De plus, sans cette hypothèse l'algorithme de Knuth-Morris-Pratt serait soit incorrect soit non terminal.

**Algorithm 7** Algorithme de Knuth–Morris–Pratt.

```
1:  function KMP(T, P)  ▷ T texte ; P motif
2:      t ← T.longueur
3:      p ← P.longueur
4:      i ← 1
5:      j ← 1
6:      s ← SUPPLÉANCE(x, P)
7:      while i ≤ p et j ≤ t do
8:          if i ≥ 1 et t[j] ≠ x[i] then
9:              i ← s[i]
10:         else
11:             i ← i + 1
12:             j ← j + 1
13:         end if
14:     end while
15:     if then i > m
16:         Renvoie j − m
17:     else
18:         Renvoie 0
19:     end if
20: end function
```

*Remarque* : La validité de l'algorithme de Knuth–Morris–Pratt (Algorithme 7) provient des propriétés sur le bord.
*Complexité du prétraitement Calcul de la fonction suppléance s* : $\Theta(p)$ On applique une méthode de l'agrégat : dans la boucle pour la variable ne peut augmenter plus de $p$ fois et comme elle est toujours strictement positive, la boucle tant que ne peut pas s'exécuter plus de $p$ fois.
*Complexité* : $\Theta(t)$ On applique une méthode de l'agrégat (exactement le même raisonnement).
*Remarque* : Il existe encore une version améliorer de cet algorithme qui consiste à ne pas vouloir se retrouver dans la même situation qu'au début (on teste toujours une situation différente). Cette amélioration donne les même complexité asymptotique mais semblerait plus rapide en pratique.

**Algorithme de Boyer et Moore [1, p.358]** *Principe* : on fait la vérification du motif de la droite vers la gauche. En cas d'erreur et si le caractère d'erreur n'est pas dans le motif, on décale le motif de $p$ caractères dans le texte. En effet, si une erreur de ce type apparaît lors de la vérification, on sait que le motif n'est pas dans le texte vérifié.

*Exemples*

| Cas 1 | Cas 2 | Cas 3 | Cas 4 |
|---|---|---|---|
| stupid_spring_string | stupid_spring_string | stupid_spring_string | stupid_spring_string |
| string | . . . . . string | . . . . . string | . . . . . string |

Dans le cas 1 et 3 : le caractère d'erreur ne se trouve pas dans le motif. On peut sauter directement dans le texte les 6 caractères du motif et recommencer par la fin du motif.

Dans le cas 2 et 4 : le g de la clé ne correspond pas au n du texte. Cependant, le motif contient un n, un caractère avant, on peut donc décaler d'une position, et vérifier à nouveau en commençant par la fin du motif.

*Complexité* : $O(pt)$

**Arbre des suffixe**

## Distance d'édition

Les distances d'éditions permettent de donner la similitude entre deux mots. On définit ici ce que sont ces distances d'éditions et quelques une de leur propriétés.

**Distance d'édition** On peut définir plusieurs distances sur les mots (plus ou moins pratique) et plusieurs distances d'éditions. Nous allons en étudier quelques unes. Nous allons commen-

cer par définir des distances théoriques (elles n'ont aucune utilité en pratique mais peuvent servir pour des raisonnements).

— La distance préfixe : $\forall u, v \in \Sigma^*$, $d_{pref} = |u| + |v| - 2|lcp(u,v)|$ où $lcp(u,v)$ est le plus long préfixe commun de $u$ et $v$.

— La distance suffixe : $\forall u, v \in \Sigma^*$, $d_{suff} = |u| + |v| - 2|lcs(u,v)|$ où $lcs(u,v)$ est le plus long suffixe commun de $u$ et $v$.

— La distance facteur : $\forall u, v \in \Sigma^*$, $d_{fact} = |u| + |v| - 2|lcp(u,v)|$ où $lcf(u,v)$ est le plus long facteur commun de $u$ et $v$.

Une distance plus importante et surtout utilisé en pratique est la distance de Hamming. Elle n'est pas très expressive : il y a une hypothèse forte sur la longueur des mots et elle nous donne juste un nombre et non quelles sont les modifications pour arriver à ce nombre. Cependant, elle reste la plus facile à calculer (linéaire en la taille du mot), c'est probablement pour cela qu'elle reste si utilisée.

**Définition.** La distance de Hamming sur deux mots de la même longueur donne le nombre de différences de lettres entre les deux mots.

*Exemple* : Si $u = aab$ et $v = abb$, la distance de Hamming appliquée à $u$ et à $v$ est deux.

La distance d'édition est une distance définie sur toutes les paires de mots. De plus, contrairement à la distance de Hamming, on est capable de donner une explication du passage de $x$ à $y$. Pour cela, nous commençons par définir trois opérations élémentaires munies de leurs coûts.

— La substitution qui positionne $a$ à la place de $b$ dans $x$ à une position donnée. On note sa fonction coût $Sub(a,b)$.

— L'insertion qui insère $a$ dans $x$ à une position donnée. On note sa fonction coût $Ins(a)$.

— La suppression qui supprime $a$ d'une position donnée dans $x$. On note sa fonction coût $Del(a)$.

**Définition.** La distance d'édition de Leveinstein est une fonction $Dist$ définie telle que :

$$Dist(x,y) = \min\{\text{coût de } \sigma : \sigma \in \Sigma_{x,y}\}$$

où $\Sigma_{x,y}$ est une suite d'opérations élémentaires permettant de transformer $x$ en $y$ où son coût est la somme des coûts de ces opérations élémentaires.

*Remarque.* La distance de Hamming est une distance d'édition où $Del(a) = Ins(a) = +\infty$.

**Proposition.** *$Dist$ est une distance définie sur $\Sigma^*$ (au sens mathématiques) si et seulement si $Sub$ est une distance sur $\Sigma$ et si $Del(a) = Ins(a) > 0, \forall a \in \Sigma$.*

*Démonstration.* $\Rightarrow$ Supposons que $Dist$ est une distance.

— $\forall a, b \in \Sigma, Dist(a,b) = Sub(a,b)$ : $Sub$ est une distance.

— $Del(a) = Dist(a,\epsilon) = Dist(\epsilon,a) = Ins(a)$ : $Del(a) = Ins(a) > 0$.

$\Leftarrow$ Montrons que $Dist$ est une distance : on montre qu'elle possède les quatre propriétés.

— Positivité : $Sub \geq 0$ ($Sub$ distance), $Del, Ins > 0$ (hypothèse) : leur somme $\geq 0$.

— Séparation : si $u = v$, $Dist(u,v) = 0$ ($Sub$ distance). Réciproquement, si $Dist(u,v) = 0$, $u = v$ ($Sub$ est la seule fonction qui s'annule).

— Symétrie : la fonction $Sub$ est symétrique ($Sub$ distance) et les fonctions $Ins$ et $Del$ sont équivalentes : $Dist$ est symétrique (suite minimale est donnée par lecture inverse).

— Inégalité triangulaire : raisonnement par l'absurde, on suppose que $\exists w \in \Sigma^*$ tel que $Dist(u,w) + Dist(w,v) < Dist(u,v)$. On a donc une suite minimale de $u$ à $w$ puis une de $w$ à $v$ de coût inférieur à une suite minimale de $u$ à $v$.

□

*Problèmes pour le calcul* : Il n'existe pas d'unicité du calcul et cela reste un problème d'optimisation.

**Alignement** L'alignement est une façon de visualiser leur similitudes. Lors du calcul de la distance d'édition (*Dist*), nous calculons un alignement optimal. Un alignement entre deux mots $x$ et $y$ est un mot $z \in (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \setminus \{(\epsilon, \epsilon)\}$ dont la projection sur la première composante est $x$ et la projection sur la seconde est $y$.

*Exemple* : Voici un alignement (qui est même optimal).

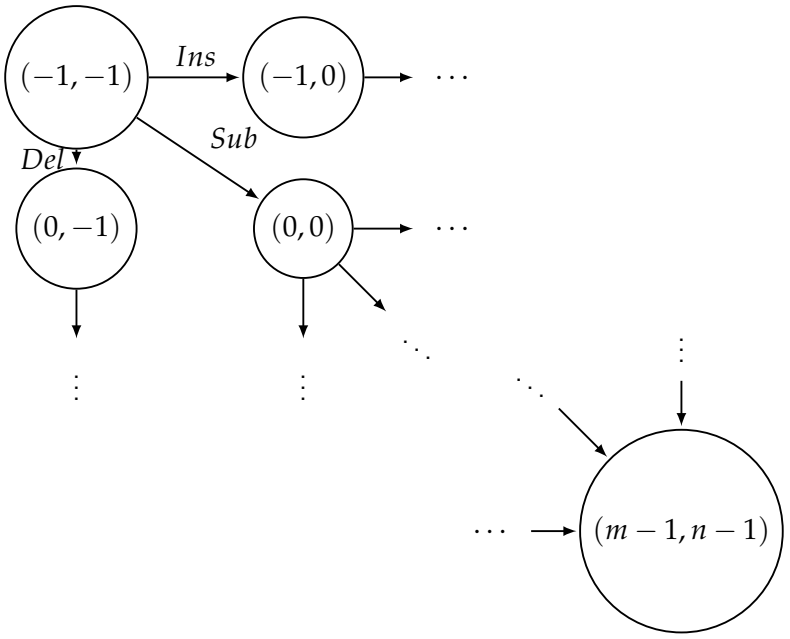
$$\begin{pmatrix} A & C & G & - & - & A \\ A & C & G & C & T & A \end{pmatrix}$$

Le nombre d'alignements entre deux mots est exponentiel. On donne maintenant une borne sur ce nombre.

**Proposition.** *Soient $x, y \in \Sigma^*$ de longueurs respectives $m$ et $n$ avec $m \leq n$. Le nombre d'alignement entre $x$ et $y$ ne contenant pas de suppression consécutives de lettres de $x$ est $\binom{2n+1}{m}$.*

*Démonstration.* Un alignement est caractérisé par les substitutions aux $n$ positions sur $y$ ainsi que les suppressions ($n+1$ emplacements possibles (en comptant un emplacement avant $y[0]$ et un après $y[n-1]$)) qui le compose. Un alignement est donc un choix de $m$ substitutions ou suppressions parmi $2n+1$ emplacements possibles. □

**Graphe d'édition** Le graphe d'édition traduit l'ensemble des choix qui forment l'alignement (Figure 1).




Le graphe d'édition est composé des sommets correspondant à l'ensembles des paires de préfixes des deux mots (un chemin de l'origine jusqu'à un nœud traduit la transformation d'un préfixe vers le second). On passe alors d'un état vers un autre à l'aide d'une opération. Tout chemin de $(-1,-1)$ vers $(m-1, n-1)$ dans ce graphe est un alignement entre $x$ et $y$.

FIGURE 1 – Un exemple partiel de graphe d'édition.

Le calcul d'un alignement optimal ou de *Dist* se ramène au calcul d'un chemin de coût optimal sur le graphe d'édition $G$. Comme $G$ est acyclique, on a besoin d'un seul passage pour le calculer (on met un ordre topologique). On utilise alors la programmation dynamique afin de calculer le graphe et d'en faire le parcours.

### Algorithme CYK

L'algorithme CYK (Cocke–Younger–Kasami) [2, p.198] est un algorithme qui décide en temps cubique si un mot est engendré par une grammaire en forme normale quadratique. Il donne alors un certificat que tout langage algébrique est dans la classe P : comme toute grammaire algébrique est équivalente à une grammaire en forme normale quadratique, tout langage algébrique peut être décidé en temps cubique. Il faut faire attention au fait que la grammaire est fixée et qu'elle ne fait pas partie de l'entrée car la mise en forme normale d'une grammaire algébrique peut être exponentielle en sa taille.

**Définition.** Le problème du mot pour une grammaire algébrique :
entrée : une grammaire algébrique $G = (\Sigma, T, R)$, $w \in \Sigma^*$ un mot
sortie : oui si $w \in L_G$ le langage engendré par $G$ ; non sinon

*Remarque* : En général répondre à ce problème est coûteux $O(|w|^3|G|^{|w|})$.

**Définition.** Soit $G$ une grammaire algébrique. On dit que :

— $G$ est sous forme normale si les règles sont sous la forme $A \to BCD\dots Z$ où $A \in T$ (alphabet de travail) et $B, \dots, Z \in \Sigma \cup T$.

— $G$ est sous forme normale de Chomsky si les règles sont sous la forme : $A \to \alpha$ pour, $A \in V, \alpha \in \Sigma$ $V \cap \Sigma = \emptyset$ ; $S \to \epsilon$ et $A \to BC$ pour $A \in V$ et $B, C \in V \setminus \{S\}$

**Théorème.** *Le problème du mot pour une grammaire algébrique sous forme forme normale de Chomsky est décidable et un algorithme de programmation dynamique le décide en temps $O(|w|^3|G|)$ et en mémoire $O(|w|^3)$.*

*Démonstration.* Cet algorithme est un algorithme de programmation dynamique. Soit $w = a_1 \dots a_n$ un mot.

On note :

— $1 \leq i \leq j \leq n$, $w[i,j] = a_i \dots a_j$

— $1 \leq i \leq j \leq n$,$E_{i,j} = \{S$ des variables telles que$w[i,j] \in L_G(S)\}$. L'algorithme calcul tous ces $E_{i,j}$.

Nous allons maintenant énoncer quelques propriétés d'appartenance à ces ensembles $E_{i,j}$ :

— $w \in L_G(S_0)$ si $S_0 \in E_{1,n}$

— $S$ appartient à $E_{i,i}$ si et seulement si $S \to a_i$ est une règle de $G$ ($w[i,i] = a_i$ et on a une grammaire en forme normale quadratique)

— $S$ appartient à $E_{i,j}$ $(i < j)$ si et seulement s'il existe une règle $S \to S_1 S_2$ et $k \in \mathbb{N}$ tels que $w[i,k] \in L_G(S_1)$ et $w[k+1,j] \in L_G(S_2)$.

Les ensembles $E_{i,j}$ peuvent être calculés à partir des ensembles $E_{i,k}$ et $E_{k+1,j}$ pour $i \leq k < j$. L'algorithme les calcul par récurrence sur la différence $j - i$.

**Algorithm 8** Algorithme CYk (Cocke–Younger–Kasami).

```
1:  function CYK(w)                          ▷ w = a_1 ... a_n : mot
2:      for 1 ≤ i ≤ j ≤ n do                 ▷ Initialisation ;
        Complexité : O(|w|^2) (double boucle)
3:          E_{i,j} ← ∅
4:      end for
5:      for i = 1 à n do                     ▷ Cas i = j ; Complexité :
        O(|w||G|) (double boucle : une sur w, une sur G)
6:          for tout règle S → a do
7:              if a_i = a then
8:                  E_{i,i} ← E_{i,i} ∪ {a}
9:              end if
10:         end for
11:     end for
12:     for d = 1 à n do                     ▷ Cas i < j ; d = j − i ;
        Complexité : O(|w|^3|G|) (triple boucle sur w ; une sur G)
13:         for i = 1 à n − d do
14:             for k = i à i + d do
15:                 for tout règle S → S_1 S_2 do
16:                     if S_1 ∈ E_{i,k} et S_2 ∈ E_{k+1,i+d} then
17:                         E_{i,i+d} ← E_{i,i+d} ∪ {S}
18:                     end if
19:                 end for
20:             end for
21:         end for
22:     end for
23: end function
```

*Complexité* : Comme la taille de la grammaire est fixé et ne fait pas partie de l'entrée (c'est une constante), on a bien la complexité annoncé en $O(|w|^3)$ □

## Références


[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] O. Carton. *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[4] M. Crochemore, C. Hancart, and L. Lecroq. *Algorithmique du texte*. Vuibert, 2001.

[5] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1994.