ALGORITHMES DU TEXTE

Résumé de cours

Professeur Christian MICHEL

Université de Strasbourg
Département d'Informatique

c.michel@unistra.fr
https://dpt-info.di.unistra.fr/~c.michel/

PLAN

1. DEFINITIONS

1.1. ALPHABET ET MOTS

Un **alphabet** A est un ensemble fini non vide dont les éléments sont appelés des **lettres**.

Un **mot** x sur un alphabet A est une suite finie d'éléments de A. La suite de zéro lettre est appelée le **mot vide** et noté ε.

L'ensemble de tous les mots sur l'alphabet A est noté A*.

L'ensemble de tous les mots sur l'alphabet A excepté le mot vide ε est noté A^+

$$A^+ = A^* \setminus \varepsilon$$

La **longueur** d'un mot x est la longueur de la suite associée au mot x et est notée |x|.

On note x[i], avec $0 \leq i \leq |x|-1$, la lettre à l'indice i de x avec par convention une numérotation des indices à partir de 0. L'indice i représente une **position** sur x si $x \neq \varepsilon$. La j-ième lettre de x est la lettre à la position j-1 sur x et x = x[0] x[1] … x[|x|-1].

L'**égalité** de 2 mots quelconques x et y est définie par

$$x = y \text{ si et seulement si } |x| = |y| \text{ et } x[i] = y[i] \text{ pour } 0 \leq i \leq |x|-1$$

L'ensemble des lettres sur lequel est formé le mot x est noté alph(x).

Le **produit** (ou **concaténation**) de 2 mots x et y est le mot composé des lettres de x puis de celles de y dans cet ordre. Ce produit est noté xy ou x·y (pour faire apparaître une décomposition du mot résultant).

L'élément neutre pour le produit est ε.

Pour tout mot x et tout entier $n \geq 0$, la n-ième **puissance** du mot x notée x^n est définie par

$$x^0 = \varepsilon$$

$$x^k = x^{k-1}\, x \text{ avec } 1 \leq k \leq n$$

Sont notés respectivement zy^{-1} et $x^{-1}z$ les mots x et y lorsque z = xy.

Le **renversé** (ou **image miroir**) du mot x est le mot $x^{\sim}$ défini par

$$x^{\sim} = x[|x| - 1]\, x[|x| - 2] \ldots x[0]$$

Un mot x est un **facteur** d'un mot y s'il existe 2 mots u et v tels que

y = u x v

Lorsque $u = \varepsilon$, x est **un préfixe** de y.

Lorsque $v = \varepsilon$, x est **un suffixe** de y.

Un facteur, préfixe ou suffixe x d'un mot y est qualifié de **propre** si $x \neq y$.

Une occurrence d'un mot non vide x dans un mot y, apparaît lorsque x est un facteur de y. Une occurrence de x peut être caractérisée par une position sur y. Une occurrence de x **débute** à la **position** (gauche) i sur y lorsque

$$y[i \,..\, i + |x|-1] = x$$

Il est également possible de considérer la **position droite** i + |x|-1 où l'occurrence se termine.

Les notations avec des crochets pour les lettres des mots sont étendues aux facteurs. On définit le facteur x[i .. j] du mot x par

$$x[i .. j] = x[i]\, x[i + 1] \ldots x[j]$$

pour tous entiers i et j satisfaisant $0 \le i \le |x|$, $-1 \le j \le |x| - 1$ et $i \le j + 1$.

Lorsque $i = j + 1$, le mot x[i .. j] est le mot vide.

1.2. LANGAGES

Tout sous-ensemble de A* est **un langage** X sur l'alphabet A.

Le produit défini sur les mots est étendu aux langages en posant $XY = X \cdot Y = \{xy : (x,y) \in X \times Y\}$ pour tous langages X et Y.

Similairement, la puissance définie sur les mots est étendue aux langages en posant $X^0 = \{\varepsilon\}$ et $X^k = X^{k-1} X$ avec $k \ge 1$.

L'étoile de X est le langage $X^* = \bigcup_{n \ge 0} X^n$.

Le langage X^+ est défini par $X^+ = \bigcup_{n \ge 1} X^n$.

La longueur définie sur les mots est étendue aux langages en posant $|X| = \sum_{x \in X} |x|$.

Les ensembles des facteurs, des préfixes et des suffixes des mots d'un langage X sont des langages particuliers notés respectivement Fact(X), Préf(X) et Suff(X).

1.3. PERIODICITE ET BORDS

Soit x un mot non vide. Un entier p tel que $0 < p \le |x|$ est une période de x si

$$x[i] = x[i + p] \text{ pour } 0 \le i \le |x| - p - 1$$

La longueur d'un mot non vide est une période de ce mot, de telle sorte que tout mot non vide possède au moins une période. Ainsi, la **période** pér(x) d'un mot non vide x est la plus petite de ses périodes.

Un **bord** d'un mot x est un facteur propre de x qui est à la fois préfixe et suffixe de x.

La fonction Bord: $A^* \to A^*$ définie pour tout mot non vide x par Bord(x) est le plus long des bords de x. On dit de Bord(x) qu'il est le **bord** de x.

Le bord de tout mot de longueur 1 est le mot vide.

Le bord d'un bord d'un mot x donné est aussi un bord de x.

Proposition

Soient x un mot non vide et n le plus grand des entiers k pour lequel $\text{Bord}^k(x)$ est défini (soit $\text{Bord}^n(x)=\varepsilon$). Alors

$$\langle \text{Bord}(x), \text{Bord}^2(x), \ldots, \text{Bord}^n(x) \rangle$$

est la suite des bords de x classés par ordre décroissant de longueur, et

$$\langle |x| - |\text{Bord}(x)|, |x| - |\text{Bord}^2(x)|, \ldots, |x| - |\text{Bord}^n(x)| \rangle$$

est la suite des périodes de x classées par ordre croissant.

1.4. PUISSANCE ET PRIMITIVITE

Proposition
Soient x et y 2 mots. S'il existe 2 entiers naturels non nuls m et n tels que $x^m = y^n$, x et y sont des puissances d'un certain mot z.

Un mot non vide est **primitif** s'il n'est puissance d'aucun autre mot que lui-même. Autrement dit, un mot $x \in A^+$ est primitif si et seulement si la décomposition $x = u^n$ avec $u \in A^*$ et $n \in N$ implique n = 1.

Proposition
Un mot non vide est primitif si et seulement s'il n'est un facteur de son carré qu'en tant que préfixe et suffixe.

Deux mots x et y sont **conjugués** s'il existe 2 mots u et v tels que x = uv et y = vu. Deux mots conjugués ont même longueur. La conjugaison est une relation d'équivalence. Elle n'est pas compatible avec le produit.

Proposition
Deux mots non vides x et y sont conjugués si et seulement s'il existe un mot z tel que xz = zy.

2. MOTS PARTICULIERS

2.1. LES MOTS DE FIBONACCI

2.1.1. Définition

Les nombres de Fibonacci sont définis par la récurrence

$$F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2$$

avec $F_0 = 0$ et $F_1 = 1$.

Propriétés remarquables

(i) $\forall n \geq 3$, $p\gcd(F_n, F_{n-1}) = 1$.

(ii) $\forall n \geq 0$, F_n est l'entier le plus proche de $\Phi^n / \sqrt{5}$ où $\Phi = \left(1+\sqrt{5}\right)/2$ (nombre d'or).

Les mots de Fibonacci sont définis sur l'alphabet $A = \{a,b\}$ par la récurrence

$$f_n = f_{n-1} \cdot f_{n-2} \text{ pour } n \geq 3$$

avec $f_0 = \varepsilon$, $f_1 = b$ et $f_2 = a$.

La suite des longueurs des mots Fibonacci est exactement la suite des nombres de Fibonacci.

2.1.2. Les 9 premiers nombres et mots de Fibonacci

2.1.3. Morphisme de Fibonacci

Un morphisme f de A^* est une application de A^* dans A^* qui satisfait

$$f(x \cdot y) = f(x) \cdot f(y) \text{ pour } x, y \in A^*$$

avec $f(\varepsilon) = \varepsilon$.

Pour tout entier positif n et tout mot $x \in A^*$, on note $f^n(x)$ le mot défini par

$$f^k(x) = f^{k-1}(f(x)) \text{ pour } k = 1,2,...,n$$

avec $f^0(x) = \varepsilon$.

Soit l'alphabet $A = \{a,b\}$. Soit φ le morphisme sur A^* défini par

$$\varphi(a) = ab \text{ et } \varphi(b) = a$$

alors le mot $\varphi^n(a)$ est le mot de Fibonacci d'ordre $n+2$ pour $n \geq 1$.

2.1.4. Palindrome de Fibonacci

Un mot u ∈ A* est un palindrome si

$$u = \varepsilon \text{ ou } u = u_1 \ldots u_n = u_n \ldots u_1$$

avec n = |u| et $u_i \in A$, $1 \leq i \leq n$.

Rappel

Les mots de Fibonacci sont définis sur l'alphabet $A = \{a,b\}$ par la récurrence

$$f_n = f_{n-1} \cdot f_{n-2} \text{ pour } n \geq 3$$

avec $f_0 = \varepsilon$, $f_1 = b$ et $f_2 = a$.

Alors pour $i \geq 2$

$$f_i = \begin{cases} uab & si\ i\ impair \\ uba & si\ i\ pair \end{cases}$$

où u est un palindrome.

2.2. LES MOTS DE DE BRUIJN

2.2.1. Définition

Les mots de de Bruijn considérés ici sont définis sur l'alphabet $A = \{a,b\}$ et sont paramétrés par un naturel non nul. Un mot non vide $x \in A^+$ est un mot de de **Bruijn** d'ordre k si chaque mot sur A de longueur k apparaît une fois et une seule dans x.

2.2.2. Exemples

2.2.3. Automate associé à un mot de de Bruijn

L'existence d'un mot de de Bruijn d'ordre $k \geq 2$ se vérifie à l'aide de l'automate défini de la façon suivante

- les états sont les mots du langage A^{k-1}
- les flèches sont de la forme (av, b, vb) avec $a,b \in A$ et $v \in A^{k-2}$

L'état initial et les états terminaux ne sont pas précisés.

3. ALIGNEMENTS DE MOTS

3.1. ALGORITHMES DE RECHERCHE DE MOTIFS

3.1.1. Introduction

Un **motif** représente un langage non vide ne contenant pas le mot vide, c.-à-d. un ensemble fini de mots éventuellement restreint à un mot.

Les algorithmes de **recherche de motifs** ont pour objectif de localiser les occurrences des mots du langage dans un autre mot appelé texte.

L'**entrée** d'un algorithme de recherche de motifs peut être

- un mot x, ou un langage X, associé ou non à sa longueur
- un texte y associé ou non à sa longueur

La **sortie** d'un algorithme de recherche de motifs peut être

- une valeur booléenne qui teste la présence ou non du mot x dans le texte y sans préciser les positions des occurrences
- un mot y' sur l'alphabet {0,1} qui code l'existence des positions droites d'occurrence de x dans y défini de la façon suivante

$$|y'| = |y| \text{ et } y'[j] = 1$$

SSI j est la position droite d'une occurrence de x dans y

- un ensemble de positions droites ou gauches donnant les occurrences de x dans y.

3.1.2. Notion de fenêtre glissante

La recherche d'un **mot x de longueur m** dans un **texte y de longueur n** se réalise au moyen d'une **fenêtre glissante**. La fenêtre délimite un facteur sur y appelé **contenu** de la fenêtre. Elle glisse le long de y de la gauche vers la droite.

La fenêtre étant à une position j donnée sur y, l'algorithme teste si x apparaît ou non (**tentative**) en j en comparant certaines lettres du contenu de la fenêtre (y) avec les lettres de x alignées en correspondance. L'occurrence de x est signalée si la tentative est vérifiée.

Lors de cette phase de test, l'algorithme acquiert une certaine information sur le texte qui peut être exploitée de 2 façons

(i) pour déterminer la longueur du prochain **décalage** de la fenêtre selon des règles propres à l'algorithme.

(ii) pour éviter des tentatives suivantes inutiles en mémorisant de l'information.

Un **décalage de longueur d** fait passer la fenêtre de la position j à la position j + d, $d \geq 1$. Un décalage doit être **valide**, c.-à-d. que pour $d \geq 2$, il n'existe pas d'occurrence de x entre les positions j + 1 = j + d - 1 dans y.

3.1.3. L'algorithme naïf

L'implémentation la plus simple de la fenêtre glissante est l'**algorithme dit naïf**. Elle consiste à considérer une fenêtre de longueur m qui glisse d'une position vers la droite après chaque tentative.

```
ALGORITHME_NAIF(x, m, y, n)
// la variable j correspond à la position gauche de la fenêtre sur le texte
pour j ← 0 à (n-m) faire TESTER_SI(y[j..j+m-1] = x)
```

Complexité dans le pire des cas

Le nombre de comparaisons de lettres dans le pire cas effectué par l'algorithme ALGORITHME_NAIF est O(m×n) (quadratique).
Ce cas existe lorsque x et y sont des puissances de la même lettre.

Complexité moyenne

Le nombre moyen de comparaisons de lettres effectué par l'algorithme ALGORITHME_NAIF est O(n-m) (linéaire).

3.1.4. Heuristiques

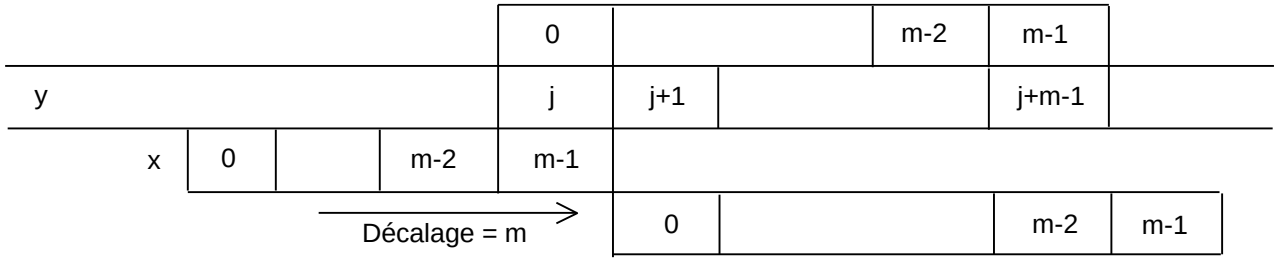
Certaines heuristiques améliorent le comportement global des algorithmes de recherche de motifs sans pouvoir démontrer une diminution de la complexité.

Une première heuristique consiste à localiser les occurrences de la première lettre x[0] du mot x dans le préfixe y[0..n-m+1] du texte y. Puis, à chaque apparition de x[0] à une position j de y, la comparaison de x[1..m-1] et y[j+1..j+m-1] est testée. Comme l'opération de recherche de l'occurrence d'une lettre est une opération de bas niveau des systèmes, le gain de temps de calcul est souvent appréciable dans la pratique.

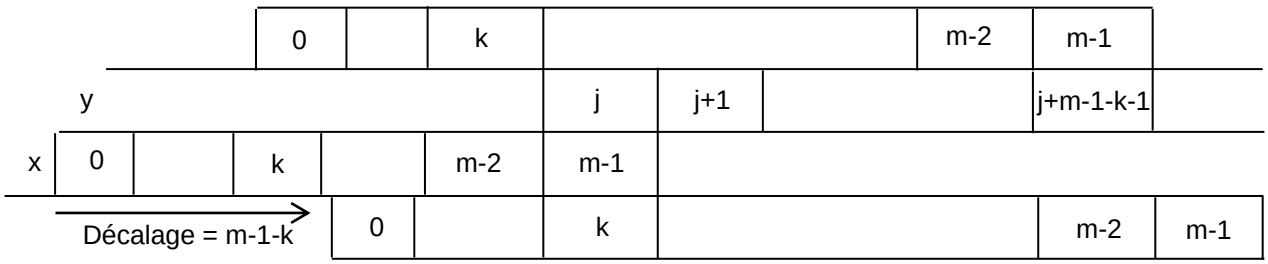
Cette recherche élémentaire peut être améliorée de 2 façons
- en positionnant la lettre x[0] de x à la suite de y
- en recherchant non plus x[0] mais la lettre de x qui a la plus faible fréquence d'apparition dans y. Mais, cette technique nécessite le calcul des fréquences des lettres de l'alphabet dans y et un calcul préalable de la position de la lettre choisie de x.

Une deuxième heuristique consiste à appliquer un décalage à la fenêtre qui tient compte de la valeur de la lettre la plus à droite dans la fenêtre. Soit j la position droite de la fenêtre. Deux cas extrêmes sont à considérer selon que la lettre y[j] apparaît ou non dans le mot x[0..m-2]

(i) Si y[j] n'apparaît pas dans x[0..m-2] alors x ne peut pas apparaître à des positions comprises entre j + 1 et j + m - 1 dans y. Un décalage de m lettres est possible.



(ii) Si k est la position maximale de y[j] dans x[0..m-2] alors x ne peut pas apparaître à des positions comprises entre j + 1 et j + m - 1 - k - 1. Un décalage de m – 2 – k + 1 = m - 1 - k lettres est possible.



Cette heuristique est codée de la façon suivante

Soit la table de la dernière occurrence dern_occ: A $\rightarrow$ {1,2,…,m} définie pour toute lettre a $\in$ A par

$$dern\_occ[a] = \min\{\{m\} \cup \{m - 1 - k: 0 \leq k \leq m - 2 \text{ et } x[k] = a\}\}$$

La table dern_occ avec dern_occ[y[j]] contient le décalage valide à appliquer après la tentative à la position droite j dans y.

```
DERNIERE_OCCURRENCE(x, m)
pour chaque lettre a ∈ A faire dern_occ[a] ← m
pour k ← 0 à m faire dern_occ[x[k]] ← m – 1 - k
retourner dern_occ
```

```
ALGORITHME_NAIF_RAPIDE(x, m, y, n)
dern_occ ← DERNIERE_OCCURRENCE(x, m)
j ← m-1
tant que (j < n) faire
    TESTER_SI(y[j-m+1..j] = x)
    j ← j + dern_occ[y[j]]
```

3.2. COMPARAISON DE MOTS

3.2.1. Distance

La notion de ressemblance ou de similarité entre 2 mots x et y de longueurs respectives m et n est duale à la notion de distance entre ces 2 mots.

Une fonction $d : A^* \times A^* \to \Re$ est une **distance** sur A^* si les 4 propriétés suivantes sont vérifiées pour tous $u,v \in A^*$

Positivité: $d(u,v) \geq 0$

Séparation: $d(u,v) = 0$ SSI $u = v$

Symétrie: $d(u,v) = d(v,u)$

Inégalité triangulaire: $d(u,v) \leq d(u,w) + d(w,v)$ pour tout $w \in A^*$

Plusieurs distances sur les mots peuvent être considérées à partir des factorisations sur les mots.

Distance préfixe: pour tous $u,v \in A^*$

$$d_{préf}(u,v) = |u| + |v| - 2 \times lpc(u,v)$$

où lpc(u,v) est le plus long préfixe commun à u et v.

Distance suffixe: distance définie symétriquement par rapport à la distance préfixe, pour tous $u,v \in A^*$

$$d_{suff}(u,v) = |u| + |v| - 2 \times lsc(u,v)$$

où lsc(u,v) est le plus long suffixe commun à u et v.

Distance facteur: distance définie de manière analogue aux 2 distances précédentes, pour tous $u,v \in A^*$

$$d_{fact}(u,v) = |u| + |v| - 2 \times lcf(u,v)$$

où lcf(u,v) est la longueur maximale des facteurs communs à u et v.

Distance de Hamming: nombre de positions entre 2 mots de même longueur avec des lettres différentes.

Distance d'édition (distance d'alignement ou distance de Levenshtein): distance la plus générale basée

(i) **la substitution** d'une lettre du mot x à une position donnée par une lettre du mot y

(ii) **la délétion** d'une lettre du mot x à une position donnée

(iii) **l'insertion** d'une lettre du mot y dans x à une position donnée

Un **coût** de valeur entière positive est associé à chacune de ces opérations. Pour $a,b \in A$

(i) Sub(a,b): coût de la substitution de la lettre a par la lettre b

(ii) Dél(a): coût de la délétion de la lettre a

(iii) Ins(b): coût de l'insertion de la lettre b

On suppose que ces coûts sont indépendants des positions auxquelles les opérations sont réalisées.

La distance de Levenshtein est définie à partir des coûts élémentaires

$$Lev(x,y) = \min\left\{ coût\ de\ \sigma : \sigma \in \sum_{x,y} \right\}$$

où $\sum_{x,y}$ est l'ensemble des suites d'opérations d'édition élémentaires permettant de transformer le mot x en un mot y et le coût $\sigma \in \sum_{x,y}$ est la somme des coûts des opérations d'édition de la suite σ.

Proposition

La fonction Lev est une distance sur A^* si

(i) Sub est une distance sur A et

(ii) $Dél(a) = Ins(a) > 0$ pour tout $a \in A$.

Remarque

La distance de Hamming est un cas particulier de distance d'édition pour laquelle seule l'opération de substitution est considérée. Elle peut être obtenue à partie de la distance d'édition en posant

$$Dél(a) = Ins(a) = +\infty$$

pour chaque lettre a de l'alphabet de 2 mots de même longueur.

Le problème du calcul de la distance de Lev(x,y) consiste à trouver une suite d'opérations d'édition élémentaires pour transformer le mot x en un mot y tout en **minimisant le coût total** des opérations utilisées, le coût total étant égal à la somme de tous les coûts associés aux opérations d'édition élémentaires.

Le problème de minimisation de la distance entre 2 mots est équivalent à maximiser la similarité entre ces 2 mots.

3.2.2. Alignements

Un **alignement** entre 2 mots $x,y \in A^*$ de longueurs respectives m et n est un mot z sur l'alphabet $\left(A \cup \{\varepsilon\}\right) \times \left(A \cup \{\varepsilon\}\right) \setminus \{\varepsilon,\varepsilon\}$ dont la projection sur la première composante est x et la projection sur la seconde composante est y.
Ainsi, un alignement z de longueur p entre x et y est noté
$z = (\bar{x}_0,\bar{y}_0)(\bar{x}_1,\bar{y}_1)...(\bar{x}_{p-1},\bar{y}_{p-1})$ avec $x = \bar{x}_0\bar{x}_1 \cdots \bar{x}_{p-1}$ et $y = \bar{y}_0\bar{y}_1 \cdots \bar{y}_{p-1}$
tel que $\bar{x}_i \in A \cup \{\varepsilon\}$ et $\bar{y}_i \in A \cup \{\varepsilon\}$ pour $i = 0,1,...,p-1$.
Il également noté

$$z = \begin{pmatrix} \bar{x}_0\bar{x}_1...\bar{x}_{p-1} \\ \bar{y}_0\bar{y}_1...\bar{y}_{p-1} \end{pmatrix}$$

Une **paire alignée** du type (a,b) avec $a,b \in A$ dénote **la substitution de la lettre a par la lettre b**.
Une paire alignée du type (a,ε) avec $a \in A$ dénote **la suppression de la lettre a dans x**.
Une paire alignée du type (ε,b) avec $b \in A$ dénote **l'insertion de la lettre b dans y**.
Dans les alignements ou les paires alignées, le symbole "–" appelé trou se substitue classiquement au symbole ε.

Un alignement de séquences est donc une suite de mises en correspondance de lettres identiques, de lettres différentes et de lettres avec des trous. Le **coût d'un alignement** est la somme des coûts associés à chacune de ses paires alignées.
Les algorithmes d'alignement considèrent l'alignement de 2 ou plusieurs (≥ 3) séquences.

Complexité

Soient $x,y \in A$ de longueurs respectives m et n avec $m \le n$. Le nombre d'alignements entre x et y ne contenant pas de suppressions consécutives de lettres de x, est $\begin{pmatrix} 2n+1 \\ m \end{pmatrix}$ (complexité exponentielle).

3.2.3. Graphe d'édition

Un alignement se traduit en termes de **graphe d'édition** G(x, y) de 2 mots $x,y \in A^*$ de longueurs respectives m et n. Soit Q l'ensemble des sommets de G(x, y) et F son ensemble de flèches. Les flèches sont étiquetées par la fonction étiq dont les valeurs sont des paires alignées, et valuées par le coût de ces paires.

L'ensemble Q des sommets est

$$Q = \{-1,0,...,m-1\} \times \{-1,0,...,n-1\}$$

L'ensemble F des flèches est

$$\begin{aligned} F = & \{((i-1,j-1),(i,j)) : (i,j) \in Q, i \neq -1, j \neq -1\} \\ & \cup \{((i-1,j),(i,j)) : (i,j) \in Q, i \neq -1\} \\ & \cup \{((i,j-1),(i,j)) : (i,j) \in Q, j \neq -1\} \end{aligned}$$

La fonction étiq

$étiq : F \to (A \cup \{\varepsilon\}) \times (A \cup \{\varepsilon\}) \setminus \{\varepsilon,\varepsilon\}$ est définie par

$étiq((i-1,j-1),(i,j)) = (x[i],y[j])$ (**substitution**)

$étiq((i-1,j),(i,j)) = (x[i],\varepsilon)$ (**délétion dans x**)

$étiq((i,j-1),(i,j)) = (\varepsilon,y[j])$ (**insertion dans y**)

La fonction coût

$coût : F \to R$ est définie par

$coût((i-1,j-1),(i,j)) = Sub(x[i],y[j])$

$coût((i-1,j),(i,j)) = Sup(x[i])$

$coût((i,j-1),(i,j)) = Ins(y[j])$

Tout chemin du sommet initial (-1, -1) au sommet final (m-1, n-1) correspond à (est étiqueté par) un alignement entre x et y. Ainsi, le graphe d'édition G(x, y) est un automate qui reconnaît tous les alignements entre x et y. Le coût d'une flèche f de G(x, y) est le coût de son étiquette coût(étiq(f)).

Le calcul d'un alignement optimal ou de façon équivalente le calcul de la distance Lev(x,y) se ramène sur le graphe G(x, y) à la recherche d'un chemin de coût minimal de (-1, -1) à (m-1, n-1). Les chemins de coût minimal sont en bijection avec les alignements optimaux entre x et y. Puisque le graphe G(x, y) est acyclique, il est possible de trouver un chemin de coût minimal en considérant une et une seule fois chaque sommet suivant un ordre topologique:

- ligne par ligne de haut en bas et de gauche à droite à l'intérieur d'une ligne
- colonne par colonne de gauche à droite et de haut en bas à l'intérieur d'une colonne
- les antidiagonales.

Le problème se résout par **programmation dynamique**.

3.3. ALIGNEMENT OPTIMAL

La méthode de calcul d'un alignement optimal entre 2 mots utilise une technique de **programmation dynamique** qui consiste à mémoriser les résultats de calculs intermédiaires pour éviter d'avoir à les recalculer et d'éviter la complexité exponentielle.

3.3.1. Calcul de la distance d'édition

A partir de 2 mots $x, y \in A^*$ de longueurs respectives m et n, la table T à m + 1 lignes et n + 1 colonnes est définie par

$$T[i,j] = d\left(x[0..i], y[0..j]\right)$$

pour $i = -1,0,...,m-1$ et $j = -1,0,...,n-1$.

Ainsi, T[i, j] est aussi le coût minimum d'un chemin de (-1, -1) à (i, j) dans le graphe d'édition G(x, y). Le calcul de T[i, j] est basé sur la formule de récurrence suivante

Pour $i = -1,0,...,m-1$ et $j = -1,0,...,n-1$,

$$\begin{cases} T[-1,-1] = 0 \\ T[i,-1] = T[i-1,-1] + Dél\left(x[i]\right) \\ T[-1,j] = T[-1,j-1] + Ins\left(y[j]\right) \\ T[i,j] = \min\begin{cases} T[i-1,j-1] + Sub\left(x[i], y[j]\right) \\ T[i-1,j] + Dél\left(x[i]\right) \\ T[i,j-1] + Ins\left(y[j]\right) \end{cases} \end{cases}$$

La valeur à la position [i, j] dans la table T, i, j ≥ 0, ne dépend ainsi que des valeurs aux positions [i-1, j-1], [i-1, j] et [i, j-1] dans T.

T[i-1, j-1]		T[i-1, j]
	↘	↓
T[i, j-1]	→	T[i, j]

DISTANCE_EDITION(x, m, y, n) // Détermination de T[m-1, n-1] = Lev(x,y)

T[-1, -1] ← 0

pour i ← 0 à (m - 1) *faire* T[i, -1] ← T[i-1, -1] + Dél(x[i])

pour j ← 0 à (n - 1) *faire*

T[-1, j] ← T[-1, j-1] + Ins(y[j])

pour i ← 0 à (m - 1) faire $T[i,j] \leftarrow \min\begin{cases} T[i-1,j-1] + Sub\left(x[i], y[j]\right) \\ T[i-1,j] + Dél\left(x[i]\right) \\ T[i,j-1] + Ins\left(y[j]\right) \end{cases}$

Retourner T[m-1 ,n-1]

Complexité

Alors que la programmation directe de la formule de récurrence précédente conduit à une complexité exponentielle, l'algorithme DISTANCE_EDITION avec 2 mots de longueur m et n s'exécute en temps quadratique O(m×n) et en espace linéaire O(min{m,n}).

3.3.2. Calcul d'un alignement

L'algorithme DISTANCE_EDITION ne calcule que le coût de la transformation du mot x en un mot y. L'alignement optimal correspondant est obtenu en **"remontant" dans la table T depuis le sommet final [m-1, n-1] jusqu'au sommet initial [-1, -1]**. A partir d'une position [i, j], on visite parmi les 3 positions voisines [i-1, j-1], [i-1, j] et [i, j-1], l'une de celles qui a produit la valeur de T[i, j].

La validité de ce procédé peut s'expliquer avec la notion de **flèche active** dans le graphe d'édition G(x, y) qui permet d'obtenir un alignement optimal.

La flèche ((i', j'), (i, j,)) d'étiquette (a, b) est active lorsque

$$T[i,j] = T[i',j'] + Sub(a,b) \;\; si \;\; i - i' = j - j' = 1$$

$$T[i,j] = T[i',j'] + Dél(a) \;\; si \;\; i - i' = 1 \;\; et \;\; j = j'$$

$$T[i,j] = T[i',j'] + Ins(b) \;\; si \;\; i = i' \;\; et \;\; j - j' = 1$$

pour $i, i' \in \{-1,0,...,m-1\}$, $j, j' \in \{-1,0,...,n-1\}$ et $a,b \in A$.

L'étiquette d'un chemin de G(x, y) joignant (k, l) à (i, j) est un alignement optimal entre x[k..i] et y[l..j] SSI toutes ses flèches sont actives

$$Lev(x[k..i], y[l..j]) = T[i,j] - T[k,l]$$

En tout sommet du graphe d'édition, sauf en (-1,-1), entre au moins une flèche active d'après la relation de récurrence de la table T.

L'algorithme UN_ALIGNEMENT consiste ainsi à remonter le long des flèches actives jusqu'à ce que le sommet initial (-1, -1) soit atteint. La variable z de l'algorithme est un mot sur l'alphabet $\left(A \cup \{\varepsilon\}\right) \times \left(A \cup \{\varepsilon\}\right)$ et la concaténation a lieu composante par composante.

```
UN_ALIGNEMENT(x, m, y, n)
z ← (ε, ε)
(i, j) ← (m-1, n-1)
tant que ((i ≠ -1) et (j ≠ -1)) faire
    si (T[i, j] = T[i-1, j-1] + Sub(x[i], y[j])) alors
    z ← (x[i], y[j]) · z
    (i, j) ← (i-1, j-1)
    sinon
        si (T[i, j] = T[i-1, j] + Dél(x[i])) alors
            z ← (x[i], ε) · z
            i ← i - 1
        sinon
            z ← (ε, y[j]) · z
            j ← j - 1
tant que (i ≠ -1) faire
    z ← (x[i], ε) · z
    i ← i - 1
tant que (j ≠ -1) faire
    z ← (ε, y[j]) · z
    j ← j - 1
retourner z
```

Complexité

Le calcul s'exécute en temps O(m+n) et avec un espace supplémentaire en O(m+n).

Les 3 tests de validité des 3 flèches entrant dans le sommet (i, j) du graphe d'édition peuvent être effectués dans un ordre quelconque. Il existe 3! = 6 combinaisons possibles de ces 3 tests. L'algorithme UN_ALIGNEMENT privilégie le chemin avec des flèches antidiagonales. Il est possible de programmer un chemin aléatoire parmi tous les chemins possibles.

Pour calculer un alignement, il est également possible de mémoriser les flèches actives sous forme de "flèches de retour" dans une table supplémentaire lors du calcul des valeurs de la table T (non détaillé).

Le calcul d'un alignement optimal fait appel à la table T et nécessite donc un espace quadratique. Il est possible de trouver un alignement optimal en espace linéaire en utilisant une méthode "diviser pour régner" (non détaillé).

3.3.3. Calcul de tous les alignements

Tous les alignements optimaux entre les mots x et y peuvent être déterminés avec l'algorithme LES_ALIGNEMENTS basé sur le même principe de flèche active que l'algorithme UN_ALIGNEMENT. L'algorithme LES_ALIGNEMENTS fait appel à la procédure récursive ALIGNEMENT_RECURSIF pour laquelle les variables x, y et T sont globales.

LES_ALIGNEMENTS(x, m, y, n, T)

ALIGNEMENT_RECURSIF(m-1, n-1, (ε,ε))

ALIGNEMENT_RECURSIF(i, j, z)

si (i ≠- 1) *et* (j ≠- 1) *et* (T[i, j] = T[i-1, j-1] + Sub(x[i],y[j])) *alors*

A_R(i-1, j-1, (x[i], y[j]) · z)

si (i ≠ -1) *et* (T[i, j] = T[i-1, j] + Dél(x[i])) *alors* A_R(i-1, j, (x[i], ε) · z)

si (j ≠ -1) *et* (T[i, j] = T[i, j-1] + Ins(y[j])) *alors* A_R(i, j-1, (ε, y[j]) · z)

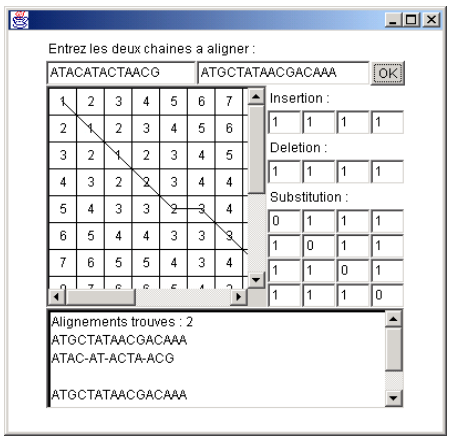
si (i = -1) *et* (j = -1) *alors* écrire que z est un alignement

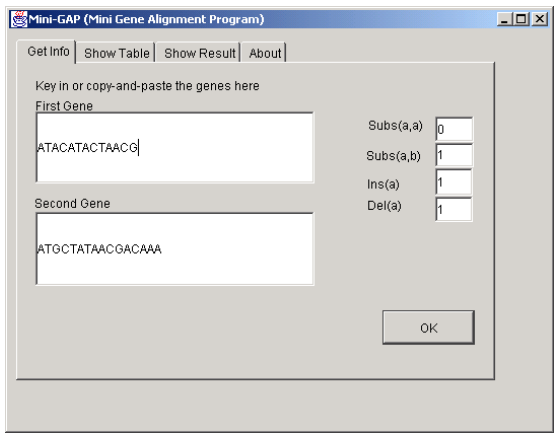
---

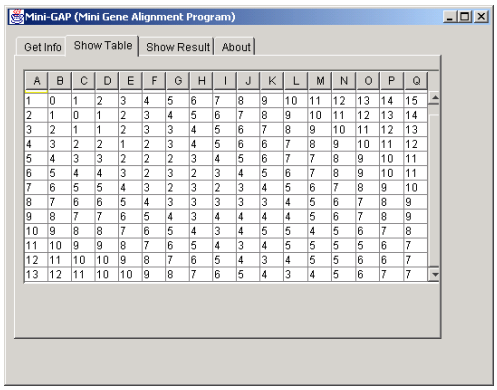
Complexité

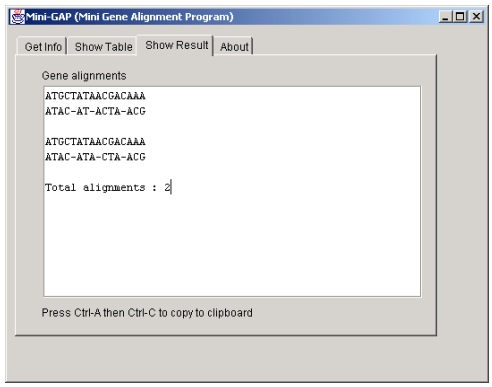
L'algorithme LES_ALIGNEMENTS a un temps d'exécution proportionnel à la somme des longueurs de tous les alignements produits.

Exemples









3.4. PLUS LONG SOUS-MOT COMMUN

Le calcul d'un plus long sous-mot commun à 2 mots x et y est **un cas particulier de la notion de distance d'édition dans laquelle l'opération de substitution n'est pas considérée.**

Deux mots x et y peuvent avoir plusieurs longs sous-mots communs.
La longueur (unique) de ces mots est notée smc(x, y).

Si $Sub(a,a) = 0$ et $Dél(a) = Ins(a) = 1$ pour $a \in A$ et
si l'on suppose $Sub(a,b) > Dél(a) + Ins(b) = 2$ pour $a,b \in A$ et $a \neq b$
alors la valeur T[m-1, n-1] représente la distance sous-mot d_{smot}(x, y) entre x et y.

Le calcul de cette distance d_{smot}(x, y) est un problème dual du calcul de la longueur smc(x, y) du plus long sous-mot commun à x et y grâce à la proposition suivante

$$d_{smot}(x,y) = |x| + |y| - 2 \times smc(x,y)$$

Une méthode naïve pour calculer la longueur smc(x,y) consiste à considérer tous 2^m sous-mots de x de longueur m, à vérifier s'ils sont des sous-mots de y et à conserver les plus longs. Cette méthode par énumération est inapplicable pour de grandes valeurs de m.

La méthode de programmation dynamique permet de calculer la longueur smc(x,y) en temps quadratique O(m×n) et en espace quadratique O(m×n). Elle calcule les longueurs des plus longs sous-mots communs à des préfixes de plus en plus longs des 2 mots x et y.
Soit la table S à m + 1 lignes et n + 1 colonnes, définie pour $i = -1,0,...,m-1$ et $j = -1,0,...,n-1$

$$S[i,j] = \begin{cases} 0 & si \;\; i = -1 \;\; ou \;\; j = -1 \\ smc\{x[0..i], y[0..j]\} & \text{sinon} \end{cases}$$

Le calcul de $smc(x,y) = S[m-1,n-1]$ est basé sur la formule de récurrence suivante

$$S[i,j] = \begin{cases} S[i-1,j-1] + 1 & si \;\; x[i] = y[j] \\ \max\{S[i-1,j], S[i,j-1]\} & \text{sinon} \end{cases}$$

pour $i = -1,0,...,m-1$ et $j = -1,0,...,n-1$.

```
SMC(x, m, y, n)   // Calcul de la longueur maximale des sous-mots communs à x et y
pour i ← -1 à (m - 1) faire S[i, -1] ← 0
pour j ← 0 à (n - 1) faire
        S[-1, j] ← 0
        pour i ← 0 à (m - 1) faire
                si (x[i] = y[j]) alors S[i, j] ← S[i-1, j-1] + 1
                sinon S[i, j] ← max{S[i-1, j], S[i, j-1]}
retourner S[m-1, n-1]
```

Complexité

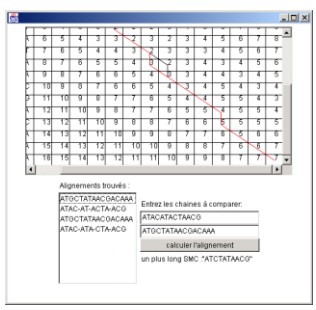
L'algorithme SMC s'exécute en temps quadratique O(m×n) et en espace quadratique O(m×n).

Un plus long sous-mot commun à x et y est obtenu en **"remontant" dans la table S depuis le sommet final [m-1, n-1] jusqu'au sommet initial [-1, -1]** (comme avec les alignements).

```
UN_PLUS_LONG_SOUS_MOT_COMMUN(x, m, y, n, S)
z ← ε
(i, j) ← (m-1, n-1)
tant que ((i ≠ -1) et (j ≠ -1)) faire
        si (x[i] = y[j]) alors
                z ← x[i] · z
                (i, j) ← (i-1, j-1)
                                                sinon
                si (S[i-1, j] > S[i, j-1]) alors i ← i - 1
                sinon j ← j - 1
retourner z
```

Il est possible de déterminer tous les plus longs sous-mots communs à x et y (comme avec les alignements).

Exemple



4. RECHERCHE APPROCHEE DE MOTS

La recherche approchée consiste à localiser toutes les occurrences des facteurs d'un texte y qui sont à distance au plus d'un entier k du mot x, $k < m \leq n$.

4.1. RECHERCHE DE MOTS A JOKERS DANS LE TEXTE ET DANS LE MOT

Le mot x et le texte y peuvent contenir des occurrences de la lettre § dite joker, lettre spéciale qui n'appartient pas à l'alphabet A. Le joker s'apparie à lui-même ainsi qu'à toutes les lettres de l'alphabet A.

La notion de **correspondance** sur $A \cup \{\S\}$ est introduit de la façon suivante:

Deux lettres a et b sur l'alphabet $A \cup \{\S\}$ se correspondent, noté $a \approx b$, si elles sont égales ou si l'une d'entre elles au moins est le joker.

Cette notion de correspondance est étendue aux mots: 2 mots u et v sur l'alphabet $A \cup \{\S\}$ et de même longueur m se correspondent, noté $u \approx v$, si à chaque position leurs lettres respectives se correspondent

$$u[i] \approx v[i] \text{ pour } i = 0,1,...,m-1.$$

La recherche de toutes les occurrences d'un mot x à jokers de longueur m dans un texte y de longueur n consiste à détecter toutes les positions j sur y pour lesquelles $x \approx y[j .. j + m - 1]$.

Le problème de la localisation d'un mot x dans un texte y lorsqu'ils contiennent des jokers ne peut se résoudre avec la notion de correspondance. En effet, la relation ≈ n'est pas transitive: pour $a, b \in A$, les relations $a \approx \S$ et $\S \approx b$ n'impliquent pas nécessairement $a \approx b$.

Il existe une méthode permettant de localiser toutes les occurrences d'un mot x à jokers de longueur m dans un texte y à jokers de longueur n ($n \geq m \geq 1$) basée sur les vecteurs binaires.

On introduit l'opérateur binaire $\otimes$. Pour des vecteurs binaires p et q d'au moins un bit, $p \otimes q$ est le produit de p et q qui est un vecteur de |p| + |q| - 1 bits (m+n-1) défini par

$$(p \otimes q)[l] = \bigvee_{i+j=l} p[i] \wedge q[j]$$

pour $l = 0,1,...,|p| + |q| - 2$.

On introduit le vecteur caractéristique λ. Pour tout mot $u \in A \cup \{\S\}$ et toute lettre $a \in A$, λ(u, a) est le vecteur caractéristique des positions de a sur u défini de longueur |u| bits satisfaisant

$$\lambda(u,a)[i] = \begin{cases} 1 & si\ u[i] = a \\ 0 & sinon \end{cases}$$

pour $i = 0,1,...,|u| - 1$.

Proposition

Si r est le vecteur de m + n - 1 bits tel que

$$r = \bigvee_{a,b \in A \text{ et } a \neq b} \lambda(y,a) \otimes \lambda(x\sim,b)$$

où $x\sim$ est le renversé de x: $x\sim = x[|x|-1]x[|x|-2]...x[0]$, alors

x ≈ y[l - m + 1 .. l] SSI r[l] = 0

pour $l = m-1, m, ..., n-1$.

4.2. RECHERCHE APPROCHEE AVEC DIFFERENCES

La recherche approchée avec k différences consiste à trouver tous les facteurs de y qui sont à une distance maximale k donnée de x, m = Ixl, n = Iyl avec k ∈ N et k <m ≤ n. La distance d'édition classique entre 2 mots x et y qui est basée sur le coût minimum des opérations d'édition élémentaires entre ces 2 mots, est prolongée avec une notion restreinte de distance obtenue en considérant le nombre minimal d'opérations d'édition (différences de type substitutions, délétions et insertions) plutôt que la somme de leurs coûts. Cette recherche approchée peut donc être résolue par des méthodes de programmation dynamique.

4.2.1. Programmation dynamique

L'approche la plus générale est basée sur la distance d'édition ordinaire (avec des coûts des opérations d'édition non nécessairement unitaires). Le principe de recherche approchée d'un mot x avec un facteur de y revient à aligner le mot x avec un préfixe de y **en considérant que l'insertion d'un nombre quelconque de lettres de y en tête de x n'est pas pénalisant**. Avec la table T (cf. cours Alignements de mots), il suffit d'initialiser à 0 les valeurs de la première ligne de T. Les positions des occurrences de x dans y sont toutes les valeurs de la dernière ligne de T inférieures à k.

La recherche approchée avec k différences est basée sur la table R définie par

$$R[i,j] = \left\{ Lev\left(x[0..i], y[l..j]\right) : l = 0,1,...,j+1 \right\}$$

pour $i = -1, 0, ..., m-1$ et $j = -1, 0, ..., n-1$, où Lev est la distance d'édition de Levenshtein (cf. cours Alignements de mots).

Le calcul de R[i, j] est basé sur la formule de récurrence suivante

$$\begin{cases} R[-1,-1] = 0 \\ R[i,-1] = R[i-1,-1] + Dél\left(x[i]\right) \\ R[-1,j] = 0 \\ R[i,j] = \min\begin{cases} R[i-1,j-1] + Sub\left(x[i],y[j]\right) \\ R[i-1,j] + Dél\left(x[i]\right) \\ R[i,j-1] + Ins\left(y[j]\right) \end{cases} \end{cases}$$

```
DIFF(x, m, y, n, k)
1    R[-1, -1] ← 0
2    pour i ← 0 à m - 1 faire
3        R[i, -1] ← R[i-1, -1] + Dél(x[i])
4    pour j ← 0 à n - 1 faire   // boucle sur les colonnes
5        R[-1, j] ← 0
6        pour i ← 0 à m - 1 faire   // boucle sur les lignes
7            R[i,j] = min { R[i-1,j-1] + Sub(x[i],y[j])
                          { R[i-1,j] + Dél(x[i])
                          { R[i,j-1] + Ins(y[j])
8        SIGNALER_SI(R[m-1, j] ≤ k)
```

Complexité

L'algorithme DIFF(x, m, y, n, k) localise les facteurs u de y pour lesquels Lev(u, x) ≤ k (distance d'édition avec des coûts quelconques) en temps quadratique O(m×n) et en espace linéaire O(m).

4.2.2. Monotonie diagonale

Cette variante de recherche approchée avec différences est basée sur **des coûts des opérations d'édition unitaires**. Ce cas particulier permet des stratégies de calcul plus efficaces que le cas général avec l'identification d'une propriété de monotonie sur les diagonales.

En supposant que Sub(a, b) = Dél(a) = Ins(b) = 1 pour a, b ∈ A, a ≠ b, la relation de récurrence précédente se simplifie

$$\begin{cases} R[-1,-1] = 0 \\ R[i,-1] = i+1 \\ R[-1,j] = 0 \\ R[i,j] = \min \begin{cases} R[i-1,j-1] & si\ x[i] = y[j] \\ R[i-1,j-1]+1 & si\ x[i] \neq y[j] \\ R[i-1,j]+1 \\ R[i,j-1]+1 \end{cases} \end{cases}$$

pour $i = 0,1,...,m-1$ et $j = 0,1,...,n-1$.

Une diagonale d de la table R est constituée par les positions [i, j] pour lesquelles j - i = d (-m ≤ d ≤ n).

Algorithme non donné.