Algorithmique et BioInformatique

Olivier Delgrange
Université de Mons

Mai 2017

Table des matières

Avertissement

Ce document ne couvre qu'une partie de la matière enseignée dans le cours d'*Algorithmique et BioInformatique*. Le reste est disponible sur la plateforme ≪ Moodle ≫ du cours.

O. Delgrange

Introduction

Le terme *BioInformatique* désigne l'analyse de l'information génétique (séquences génétiques ou structures protéiques). Il s'agit donc d'un domaine à part entière de la biologie moléculaire et donc absolument pas d'une utilisation des outils informatiques habituels aux données et aux problèmes biologiques. Les problèmes de bioinformatique sont spécifiques et peu de méthodes informatiques existantes peuvent être directement appliquées pour résoudre ces problèmes spécifiques.

La bioinformatique n'est cependant pas une nouvelle discipline, elle est apparue, sous le nom de *BioMathématique*, dans les années 60 lors des premiers travaux de phylogénie moléculaire (reconstruction des filiations évolutives, sous forme d'arbres, d'un ensemble de séquences génétiques).

Le but de la bioinformatique est d'effectuer des analyses synthétiques des données disponibles pour proposer des hypothèses de mécanismes biologiques ou des prédictions (de gènes par exemple). C'est donc un des domaines fondateurs de la biologie moléculaire plutôt qu'un produit de celle-ci : les connaissances actuelles sur les informations de biologie moléculaire ne seraient pas à leur stade actuelle sans les études de type bioinformatique.

On peut considérer quatre aspects différents de l'apport de l'informatique pour l'étude des séquences biologiques [5] :

Compilation et organisation des données : Création et gestion des banques de données d'informations génétiques.

Traitements systématiques des séquences : Le but est de repérer ou de caractériser une fonctionnalité ou un élément biologique intéressant. Citons par exemple :
- prédiction de gènes
- prédiction des facteurs de transcription
- prédiction des introns et des exons dans les gènes
- prédiction des régions d'initiation de traduction
- recherche des similitudes entre séquences
- modélisation des réseaux de régulation de gènes
- etc

Élaboration de stratégies : Le but est d'apporter de nouvelles connaissances biologiques que l'on pourra intégrer dans des traitements standards. Citons entre autres :
- mise au point de nouvelles matrices de substitution des acides aminés
- détermination de l'angle de courbure d'un segment d'ADN étant donné sa séquence de nucléotides
- etc

Évaluation des différentes approches dans le but de les valider : les méthodes sont étroitement imbriquées pour donner naissance à un ensemble d'outils qui convergent vers un but commun d'analyse informatique des séquences.

Dans ce cours, nous nous focalisons sur un aspect précis de la bioinformatique : l'algorithmique sur les chaînes de caractères. En effet, une abstraction de base nous permet de considérer les séquences génétiques comme de simples chaînes de caractères. Dès lors, certains algorithmes classiques dans le domaine "algorithmique du texte" [3] trouvent leurs applications en

bioinformatique. Citons par exemple la recherche d'un motif dans un texte, la recherche sur base d'expressions régulières, l'indexation des segments d'un texte à l'aide de l'arbre des suffixes, ... Des problèmes plus spécifiques à la bioinformatique sont également apparus : recherche de motifs approximatifs, évaluation des ressemblances entre séquences, ...

Le premier chapitre établit un ensemble de préliminaires, d'une part dans le domaine de la biologie moléculaire, d'autre part dans le domaine de l'algorithmique du texte. Le chapitre suivant traite du problème précis de la recherche d'un motif (une séquence de caractères) dans un texte (autre séquence de caractères). Le troisième chapitre aborde le problème de la ressemblance entre séquences génétiques. Le cours se termine par un chapitre de description de l'arbre des suffixes. Il s'agit d'un index compact et très performant de tous les segments d'une séquence.

Chapitre 1

Préliminaires

1.1 Notions de biologie moléculaire

On appelle *macro-molécules* les molécules constituées d'un assemblage d'un grand nombre de molécules plus petites. Dans le cadre des macro-molécules génétiques, ils s'agit d'enchaînements linéaire de constituants plus petits. Il existe trois types de macro-molécules génétiques : les *protéines*, l'*Acide DésoxyriboNucléique (ADN)* et l'*Acide RiboNucléique (ARN)*

1.1.1 Les protéines

Les *protéines* sont les macro-molécules biologiquement actives : elles jouent un rôle particulier dans la vie de l'organisme. Les protéines sont les constituants de base de tout être vivant.

Exemple 1.1 *l'hémoglobine, les hormones de croissance, la cortisone, le prion, ...*

Une protéine est un enchaînement linéaire de molécules plus petites : les *acides aminés*. Il existe 20 acides aminés différents : l'*Alanine (Ala, A)*, la *Cystéine (Cys, C)*, l'*Acide Aspartique (Asp, D)*, [1]...

L'enchaînement étant linéaire, une protéine peut être représentée par la suite des noms des acides aminés qui la constituent. Ainsi, si l'on considère l'*alphabet des acides aminés*, $\mathcal{P} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$, une protéine est un *mot* (ou une *séquence*) sur $\mathcal{P}$. Cette chaîne de caractères est appelée la *structure primaire* de la protéine.

Exemple 1.2 $p = IIPNPACDK$ *est une protéine hypothétique*[2]*, avec* I *l'*Isoleucine*,* P *la* Proline*,* N *l'*Aspargine*,* A *l'*Alanine*,* C *la* Cystéine*,* D *l'*Acide Aspartique *et* K *la* Lysine*.*

Remarque 1.1 *En pratique, c'est la structure tridimensionnelle de la protéine qui détermine sa fonction et non la chaîne linéaire d'acides aminés. En première approximation, la structure 3D peut être complètement prédite à partir de la structure primaire de la protéine. Ce n'est en réalité pas totalement vrai, un ensemble de facteurs extérieurs influent également la forme de la protéine. Malgré tout, si deux protéines ont une structure primaire proche, elles auront une structure 3D proche et donc une fonction proche. Dès lors, nous nous limitons à la structure primaire des macro-molécules dans notre algorithmique.*

Exemple 1.3 *Visualisation (artistique) de la structure tridimentionnelle d'une protéine [6] :*

1. Chaque acide aminé possède un code à trois lettres ou un code à une lettre
2. Rien ne dit que cette molécule existe réellement chez un être vivant

1.1.2 L'ADN (*Acide DésoxyriboNucléique*)

1.1.3 L'ARN (*Acide RiboNucléique*)

1.2 Définitons et notations

Définition 1.1 *Un* alphabet $\mathcal{A}$ *est un ensemble fini de* lettres *(*caractères *ou* symboles*).*

Exemples 1.1
$\mathcal{A} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w\}$
$\mathcal{B} = \{0, 1\}$ *est l'alphabet binaire*
$\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ *est l'alphabet des nucléotides*

Définition 1.2 *Un* mot *(*chaîne *ou* séquence*) est une suite finie de lettres d'un même alphabet.*
On note $s = s_1 s_2 s_3 \dots s_n$ *un mot sur l'alphabet* $\mathcal{A}$ *si* $\forall i : 1 \leq i \leq n : s_i \in \mathcal{A}$.

Définition 1.3 *La* longueur *d'un mot* s *est son nombre de lettres. On la note* $|s|$.

Définition 1.4 *Le* mot vide *est noté* ϵ *:* $|\epsilon| = 0$.

Définition 1.5 *L'*ensemble de tous les mots *que l'on peut former à partir de l'alphabet* $\mathcal{A}$ *est noté* $\mathcal{A}^*$.

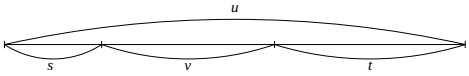
*L'*ensemble de tous les mots non vides *que l'on peut former à partir de l'alphabet* $\mathcal{A}$ *est noté* $\mathcal{A}^+ = \mathcal{A}^* \backslash \{\epsilon\}$.

Exemple 1.4 $\mathcal{B}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

Définition 1.6 *La* concaténation *de deux mots* $u, v \in \mathcal{A}^*$*, notée* $u.v$*, ou* uv*, est le mot obtenu en faisant suivre les lettres de* u *par les lettres de* v*. Par définition,* $uv \in \mathcal{A}^*$.

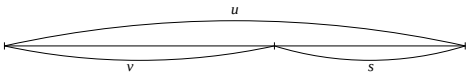
Si $u = u_1 u_2 \dots u_n$ *et* $v = v_1 v_2 \dots v_m$ *alors* $uv = u_1 u_2 \dots u_n v_1 v_2 \dots v_m$ *et* $|uv| = |u| + |v|$.

Définition 1.7 *Soit* $u \in \mathcal{A}^*$ *un mot. Un mot* $v \in \mathcal{A}^*$ *est* facteur *de* u *si et seulement si* v *est composé d'une suite de lettres consécutives de* u*. Formellement :* v *est facteur de* u *si et seulement si* $\exists s, t \in \mathcal{A}^*$ *tel que* $u = s.v.t$.

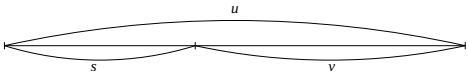


On écrit $v = u_{i..j}$ *si* $v = u_i u_{i+1} \dots u_j$.

Définition 1.8 *Un mot $v \in \mathcal{A}^*$ est* préfixe *de $u \in \mathcal{A}^*$ si et seulement si v est un facteur de u qui débute u. Formellement : v est préfixe de u si et seulement si $\exists s \in \mathcal{A}^*$ tel que $u = v.s$.*



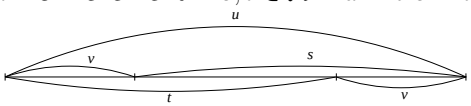
Définition 1.9 *Un mot $v \in \mathcal{A}^*$ est* suffixe *de $u \in \mathcal{A}^*$ si et seulement si v est un facteur de u qui termine u. Formellement : v est suffixe de u si et seulement si $\exists s \in \mathcal{A}^*$ tel que $u = s.v$.*



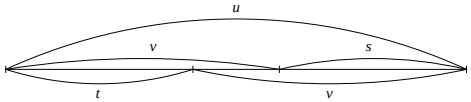
Remarque 1.2 *Un facteur, préfixe ou suffixe de u peut être le mot vide ou u lui-même.*

Définition 1.10 *Le mot v est* préfixe *(resp.* suffixe*)* propre *de u si et seulement si v est préfixe (resp. suffixe) de u et $u \neq v$.*

Définition 1.11 *Un mot $v \in \mathcal{A}^*$ est un* bord *de $u \in \mathcal{A}^*$ si et seulement si v est préfixe et suffixe de u. Formellement : $\exists s, t \in \mathcal{A}^* : u = v.s = t.v$.*



ou



Remarque 1.3 *Le mot vide ϵ est par nature un bord de tout mot.*

Définition 1.12 *Le mot $v \in \mathcal{A}^*$ est un* sous-mot *(*sous-chaîne *ou* sous-séquence*) de $u \in \mathcal{A}^*$ si et seulement si v est composé d'une suite ordonnée de lettres extraites de u. Formellement : $v = u_{i_1} u_{i_2} \dots u_{i_m}$ avec $u = u_1 u_2 \dots u_n$ et $1 \leq i_1 < i_2 < i_3 \dots < i_m \leq n$.*

Exemple 1.5 $u = \texttt{ACAAGATATAG} \in \mathcal{N}^*$. *Le mot* $v = u_2 u_5 u_6 u_8 u_{11} = \texttt{CGAAG}$ *est un sous-mot de* u.

1.3 Propriétés des bords

Les propriétés sur les bords sont primordiales en algorithmique du texte car elles permettent de découvrir rapidement certaines régularités dans les mots.

Définition 1.13 *Soit $x \in \mathcal{A}^+$ un mot sur $\mathcal{A}$. Le mot $Border(x) \in \mathcal{A}^*$ désigne le plus long bord de x qui ne soit pas x lui même.*

Exemple 1.6 *Soit* $x = \texttt{AAGAAGAA}$. *L'ensemble de tous les bords de x est* $\{\texttt{AAGAAGAA}, \texttt{AAGAA}, \texttt{AA}, \texttt{A}, \epsilon\}$ *et* $Border(x) = \texttt{AAGAA}$.

Définition 1.14 *Soit $x \in \mathcal{A}^+$ et $i > 1$ un entier. $Border^i(x)$ désigne la $i^{ème}$ itération de la fonction $Border$ au mot x : $Border^i(x) = Border(Border^{i-1}(x))$ avec $Border^1(x) = Border(x)$.*

Soit $h \geq 0$ l'unique entier tel que $Border^h(x) = \epsilon$.

Proposition 1.1 *$Border^i(x)$ est un bord de x, avec $i \leq h$.*

Preuve (récurrence) C'est vrai pour $i = 1$: $Border(x)$ est un bord de x par définition.

Supposons que ce soit vrai pour $i-1$: $Border^{i-1}(x)$ est un bord de x. Dès lors, $Border^{i-1}(x)$ est à la fois préfixe et suffixe de x.

$Border^i(x)$ est à la fois préfixe et suffixe de $Border^{i-1}(x)$ car $Border^i(x)$ est un bord de $Border^{i-1}(x)$. Par transitivité de la propriété de préfixe (resp. suffixe), $Border^i(x)$ est préfixe (resp. suffixe) de x. C'est donc un bord de x. □

Proposition 1.2 *La suite* $(x, Border(x), Border^2(x), ..., Border^h(x) = \epsilon)$ *désigne la suite, ordonnée du plus long au plus court, de tous les bords de* x*.*

Preuve

- **a.** La suite est ordonnée du plus long au plus court car $|Border^i(x)| < |Border^{i-1}(x)|$
- **b.** Si $u = Border^i(x)$, avec $0 \leq i \leq h$, alors u est un bord de x (proposition 1.1).
- **c.** C'est vrai pour $x = \epsilon$: (ϵ) est bien la liste de tous les bords de ϵ.
- **d.** Prouvons que si u est un bord de x alors $u \in (x, Border(x), Border^2(x), ..., Border^h(x))$.
  Supposons maintenant que $x \neq \epsilon$.

  Premier cas : $u = x$. Dans ce cas $u \in (x, Border(x), Border^2(x), ..., Border^h(x))$.

  Deuxième cas : $|u| < |x|$

  Preuve par récurrence : supposons que ce soit vrai pour $x' = Border(x)$:
  la suite de tous les bords de x' est $(x', Border(x'), Border^2(x'), ..., Border^{h-1}(x')) = (Border(x), Border^2(x), ..., Border^h(x))$.
  On a soit $u = x'$, dans ce cas la propriété est vraie.
  Si $u \neq x'$, u est forcément bord de x', autrement cela contredit la définition de $Border(x)$.
  Grâce à l'hypothèse de récurrence, $u \in (x, Border(x), Border^2(x), ..., Border^h(x))$. □

La proposition suivante ainsi que son corollaire permettent la construction de $Border(x.a)$, avec $a \in \mathcal{A}$ une lettre et $x \in \mathcal{A}^+$ un mot, lorsque $Border(u)$ est connu, pour tout u préfixe de x.

Proposition 1.3 *Soit* $x \in \mathcal{A}^+$ *un mot et* $a \in \mathcal{A}$*,* $Border(x.a)$ *est le plus long préfixe de* x *qui soit dans l'ensemble* $B = \{Border(x)a, Border^2(x)a, ..., Border^h(x)a, \epsilon\}$*.*

Preuve

- **a.** Soit z un bord de $x.a$. Si $z \neq \epsilon$, alors $z = z'a$ avec z' bord de x. Donc (proposition 1.2), $z' \in \{x, Border(x), Border^2(x), ..., Border^h(x)\}$ et dès lors $z \in \{x.a, Border(x)a, Border^2(x)a, ..., Border^h(x)a\}$.
  Puisque $Border(x.a)$ est un bord disjoint de $x.a$, $Border(x.a) \in B$.
- **b.** Réciproquement, tout mot $u \in B$ est suffixe de $x.a$ par construction. Un tel mot u est donc un bord de $x.a$ s'il est également préfixe de $x.a$. Puisque $Border(x.a) \neq x.a$, $Border(x.a)$ est préfixe de x. Le plus long préfixe de x présent dans B est donc $Border(x.a)$. □

Corollaire 1.4 *Soit* $x \in \mathcal{A}^+$ *un mot et* $a \in \mathcal{A}$*.*

$Border(xa) = Border(x)a$ *si* $Border(x)a$ *préfixe de* x*,*
$Border(xa) = Border(Border(x)a)$ *sinon.*

Preuve

- **a.** Si $Border(x)a$ est préfixe de x, alors il s'agit du plus long préfixe de x présent dans l'ensemble $B = \{Border(x)a, Border^2(x)a, ..., Border^h(x)a, \epsilon\}$. On a donc bien $Border(x.a) = Border(x)a$ d'après la proposition 1.3.
- **b.** Supposons que $Border(x)a$ ne soit pas préfixe de x.
  Posons $y = Border(x)$. D'après la proposition 1.3 et puisque $Border(x)a$ n'est pas préfixe de x, $Border(x.a)$ est plus court que $Border(x).a = y.a$, il est donc préfixe de y.
  Grâce à la proposition 1.3, $Border(x.a)$ est le plus long préfixe de x, donc aussi de y, dans l'ensemble $B = \{Border(x)a, Border^2(x)a, ..., Border^h(x)a, \epsilon\}$. Puisque $Border(x)a$ n'est pas préfixe de x, $Border(x.a)$ est le plus long préfixe de x présent dans $B \setminus \{Border(x)a\} = \{Border^2(x)a, ..., Border^h(x)a, \epsilon\} = \{Border(y)a, ..., Border^{h-1}(y)a, \epsilon\}$.
  Or, le plus long préfixe de y présent dans $\{Border(y)a, ..., Border^{h-1}(y)a, \epsilon\}$ est $Border(y.a)$ (application de la proposition 1.3 au mot y).
  Dès lors $Border(x.a) = Border(y.a) = Border(Border(x)a)$. □

Le corollaire 1.4 propose un mécanisme récursif de construction de $Border(x.a)$: soit $Border(x)a$ est préfixe de x et dans ce cas $Border(x.a)$ a été construit. Soit on se ramène à un problème semblable de taille plus petite : le calcul de $Border(y.a)$. Ce processus récursif possède une fin puisque $Border(a) = \epsilon$.

Chapitre 2

String Matching

2.1 Le problème

Soient $p, t \in \mathcal{A}^*$ avec $|p| = m, |t| = n$ et $m \leq n$. Le problème de *String-Matching* consiste à trouver toutes les positions h d'occurrence de p (le *motif*) dans t (le *texte*).

C'est-à-dire que l'on désire trouver toutes les positions h, avec $1 \leq h \leq n - m + 1$ telles que $p = t_{h..h+m-1}$

Exemple 2.1 *Soit* $t = \texttt{GGAGATAGAGAC} \in \mathcal{N}^*$ *et* $p = \texttt{AGA} \in \mathcal{N}^*$*. Le motif* p *est présent dans* t *aux positions* $3, 7$ *et* 9 *:*

```
123456789012
GGAGATAGAGAC
  *** ***
      ***
```

2.2 Principe de la fenêtre coulissante

Le point commun de la plupart des algorithmes de string matching est l'utilisation d'une fenêtre coulissante, de taille m, qui parcourt le texte de la gauche vers la droite. Si cette fenêtre débute à la position h, elle contient le facteur $t_{h..h+m-1}$ du texte. Les algorithmes que nous décrivons déplacent la fenêtre de la gauche vers la droite et pour une position h déterminée, déterminent si le mot qu'elle contient est p, c'est-à-dire si h est une position d'occurrence du motif. Pour cela, ils comparent les caractères de la fenêtre avec ceux qui lui correspondent dans le motif.

2.3 Complexité en temps

Nous verrons que l'opération de base des algorithmes de string matching est la comparaison de deux caractères : chaque étape procède à une comparaison, suivie d'opérations qui se déroulent en $O(1)$ (en « temps constant »). Dès lors, l'étude de la complexité en temps de l'étape de recherche réalisée par ces algorithmes consiste bien souvent à évaluer le nombre de comparaisons effectuées par l'algorithme. Les algorithmes dits *linéaires* sont ceux qui effectuent un nombre de comparaisons majoré par $K_1 \times n$ où K_1 est une constante. On écrira que la complexité en temps est en $O(n)$. Ceux qui sont *quadratiques* sont ceux dont le nombre de comparaisons est majoré par $K_2 \times n \times m$ où K_2 est une constante. On écrira que leur complexité en temps est en $O(n \times m)$. Ils sont bien entendu asymptotiquement moins efficaces que les algorithmes linéaires.

Une comparaison est d'une des deux catégories *match* ou *mismatch* :

— un *match* est une comparaison “réussie” de deux caractères : elle conclut l'identité des deux caractères.
— un *mismatch* est une comparaison “ratée” : elle conclut la différence entre les deux caractères.

Les études de complexité en temps sont souvent amenées à compter séparément le nombre de matchs et de mismatchs.

2.4 Algorithme naïf

L'algorithme naïf est également appelé “algorithme en force brute” car il effectue beaucoup de travail en essayant toutes les positions potentielles de début de la fenêtre coulissantes (c'est-à-dire toutes les positions h avec $1 \leq h \leq n - m + 1$). Pour chaque position de la fenêtre les comparaisons sont effectuées de la gauche vers la droite.

2.4.1 Première version

La première version, SM-NAÏF1, suit ce schéma. Sa complexité en temps est $O((n - m + 1)m) = O(nm)$ dans tous les cas car la boucle extérieure effectue exactement $n-m+1$ passages et chaque passage nécessite exactement m comparaisons.

```
SM-NAÏF1(p, m, t, n)
1  Pour h ← 1 Jusque n − m + 1
2      Faire match ← VRAI
3          Pour i ← 1 Jusque m
4              Faire Si p_i ≠ t_{h+i−1}
5                  Alors match ← FAUX
6          Si match
7              Alors AFFICHER(h)
```

Il paraît évident, lorsqu'un mismatch se produit, que les autres comparaisons sont inutiles dans la fenêtre courante. En effet, on peut immédiatement conclure que le motif n'est pas le mot présent dans la fenêtre. Cette constatation est valable pour plusieurs algorithmes, nous définissons donc une fonction booléenne permettant de tester la présence du motif dans une fenêtre en s'arrêtant immédiatement en cas de mismatch.

2.4.2 Fonction de test

La fonction TEST teste l'occurrence du motif à la position h du texte. Elle retourne VRAI dans le cas où le motif est présent dans le texte à la position h et FAUX dans le cas contraire. Elle réalise son travail par une comparaison, de gauche à droite des caractères de la fenêtre du texte commençant à la position h avec ceux du motif et ce jusqu'à ce que tous les caractères de la fenêtre aient étés parcourus sans mismatch ou jusqu'à ce qu'un mismatch soit rencontré. La fonction évite ainsi d'effectuer des comparaisons inutiles puisqu'un seul mismatch suffit pour conclure qu'il n'y a pas d'occurrence à cette position.

```
TEST(p, m, t, h)
1  i ← 1
2  TantQue i ≤ m cet p_i = t_{h+i−1}
3      Faire i ← i + 1
4  Si i > m
5      Alors Retourner VRAI
6      Sinon  Retourner FAUX
```

La complexité en temps dans le pire des cas de la fonction TEST est $O(m)$ car m comparaisons sont effectuées lorsque les $m-1$ premières sont des matchs.

Remarque 2.1 *L'opérateur **cet** est un opérateur **et** conditionnel : si la première condition est fausse, la deuxième n'est absolument pas considérée. Cet opérateur est non commutatif. Il est utilisé lorsque la deuxième condition n'est définie que lorsque la première est vraie. Dans beaucoup de langages de programmation, le C par exemple, l'opérateur **et** est automatiquement conditionnel*

2.4.3 Deuxième version

L'algorithme SM-NAÏF2 est une version améliorée de l'algorithme naïf utilisant la fonction TEST.

SM-NAÏF2(p, m, t, n)
1 **Pour** $h \leftarrow 1$ **Jusque** $n - m + 1$
2 **Faire Si** TEST(p, m, t, h)
3 **Alors** AFFICHER(h)

Complexité en temps

La complexité en temps dans le pire des cas est quadratique : $O(nm)$. En effet, dans le pire des cas toutes les comparaisons de la boucle intérieure sont effectuées pour chaque position de la fenêtre.

Exemple 2.2 *Soit* $t =$ `AAAAAAAAAA...A` *et* $p =$ `AA...A`*. Pour chaque position* h *de fenêtre, toutes les comparaisons doivent être effectuées.*

Considérons maintenant la complexité temporelle moyenne. Le nombre moyen de comparaisons requis dépend de la fréquence d'apparition des caractères dans le texte. Sans connaissance a priori des probabilités d'apparition, nous supposerons qu'elles sont uniformes [1].

Proposition 2.1 *Soit* $q = \#\mathcal{A}$ *la taille de l'alphabet. Si la répartition des lettres est uniforme, alors la compléxité temporelle moyenne de* SM-NAÏF2 *est* $O(\frac{q}{q-1}n)$.

Preuve. Puisque la répartition est uniforme, la probabilité qu'une comparaison soit un match est de $\frac{1}{q}$. Soit $N = n - m + 1$ le nombre de fenêtres considérées par SM-NAÏF2. La probabilité qu'une fenêtre comporte plus d'une comparaison est $\frac{1}{q}$. Le nombre moyen de comparaisons est donc $NC = N \times (1 + \frac{1}{q}NC_2)$ où :

— 1 comptabilise la première comparaison dans la fenêtre
— NC_2 comptabilise le nombre moyen de comparaisons supplémentaires à effectuer dans une fenêtre lorsque la première est un match (c'est-à-dire dans 1 cas sur q).

Le nombre NC_2 se calcule de la même manière : $NC_2 = 1+\frac{1}{q}NC_3$ car la seconde comparaison est toujours effectuée et, dans 1 cas sur q, cette deuxième comparaison est un match, il faut alors comptabiliser le nombre moyen de comparaisons supplémentaires à partir de la troisième (NC_3). De la même manière $NC_3 = 1 + \frac{1}{q}NC_4$ et ainsi de suite...

Donc $NC = N\left(1 + \frac{1}{q}\left(1 + \frac{1}{q}(1 + \frac{1}{q}(1 + ...))\right)\right) = N(1 + \frac{1}{q} + \frac{1}{q^2} + \frac{1}{q^3} + \frac{1}{q^4} + \ldots)$.

Or $1 + \frac{1}{q} + \frac{1}{q^2} + \frac{1}{q^3} + \frac{1}{q^4} + \ldots$ est une série géométrique de raison $\frac{1}{q} < 1$. Elle converge donc [2] vers $\frac{1}{1-\frac{1}{q}} = \frac{q}{q-1}$. Donc $NC = \frac{q}{q-1}N$.

1. Cette décision est raisonable dans le domaine de la bioinformatique : sans aucune connaissance d'une séquence d'ADN, on s'attend à ce qu'elle suive une distribution uniforme des nucléotides

2. La propriété de convergence concerne les séries infinies. Puisque le motif est de longueur finie, la somme contient un nombre fini de termes et la convergence est à interpréter ici comme une majoration de la somme

Le nombre moyen de comparaison est donc en $O(\frac{q}{q-1}(n-m+1)) = O(\frac{q}{q-1}n)$ □

Si l'alphabet est grand, le nombre moyen de comparaisons avoisine la longueur du texte.

Exemple 2.3 *Si l'alphabet est $\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, la complexité temporelle moyenne est $O(\frac{4}{3}n)$. Chaque symbole du texte sera donc en moyenne lu $1.33333\ldots$ fois.*

Notons enfin que dans le meilleur des cas, le nombre de comparaisons est de $n-m+1$: c'est le cas où le premier caractère du motif n'est pas présent dans le texte, la première comparaison est un mismatch pour chacune des fenêtres.

2.5 Algorithme de Karp-Rabin (1987)

Pour gagner du temps, certaines fenêtres potentielles ne vont pas être testées [4] : celles qui induisent une fenêtre du texte qui ne "ressemble pas au motif". L'aspect "ressemblance" va ici être pris en charge par une fonction de hachage. Cette fonction $hash$ va donner, pour un mot de taille m, sa signature numérique. Elle va être initialement calculée sur le motif p. Une fenêtre coulissante va parcourir toutes les positions potentielles d'occurence dans le texte. La fonction de hachage va être appliquée sur chaque fenêtre d'alignement $t_{h..h+m-1}$. Si $hash(p) = hash(t_{h..h+m-1})$, les comparaisons vont être effectuées d'une manière standard entre p et $t_{h..h+m-1}$ (utilisation de la fonction $\textsc{Test}$, voir section 2.4.2). Si la valeur de la fonction de hachage n'est pas identique, il est certain que les deux mots ne sont pas égaux. Si la valeur de la fonction de hachage est identique, cela ne signifie pas pour autant que les mots sont égaux puisqu'une fonction de hachage n'est pas forcément injective.

Les qualités attendues de la fonction de hachage sont :

— elle doit être rapide à calculer
— elle doit avoir un grand pouvoir discriminant entre les mots
— $hash(t_{h..h+m-1})$ doit être facilement obtenu à partir de $hash(t_{h-1..h+m-2})$ puisque la fenêtre glisse de position en position vers la droite.

Soit w, un mot de longueur m. Sa fonction de hachage est définie par (les lettres sont assimilées à des chiffres) :

$$hash(w_1 w_2 ... w_m) = (w_1 \times 2^{m-1} + w_2 \times 2^{m-2} + \ldots + w_m) mod\ q$$

où q est un grand nombre entier.

Dès lors, le calcul de $hash(t_{h..h+m-1})$ à partir de $hash(t_{h-1..h+m-2})$ est simplement réalisé par la formule de récurrence :

$$hash(t_{h..h+m-1}) = ((hash(t_{h-1..h+m-2}) - t_{h-1} \times d) \times 2 + t_{h+m-1}) mod\ q$$

où $d = 2^{m-1}$

Le terme correspondant au caractère qui sort est soustrait, la fenêtre est décalée vers la droite (multiplication par 2) et le caractère qui rentre dans la fenêtre est ajouté. Tout cela se passe modulo q pour respecter la formule initiale.

Remarque 2.2

— *si q est le plus grand nombre entier représentable augmenté de 1 sur notre ordinateur, le modulo est pris en charge automatiquement par le hardware. Le modulo sera donc implicite dans l'algorithme.*
— *la multiplication par 2 est très rapide en pratique sur les entiers (décalage de 1 bit), ce qui explique le choix de cette fonction de hachage.*

Après avoir calculé la valeur de la fonction de hachage sur le motif et sur la première fenêtre du texte et après avoir testé grâce à ces valeurs si le motif n'apparaît pas en première position du texte, la fenêtre commence à coulisser le long du texte. Chaque étape effectue l'actualisation de la fonction de hachage, la compare à la fonction de hachage du motif et en cas d'égalité appelle la fonction TEST qui va effectuer la comparaison de la fenêtre avec le motif.

```
SM-KR(p, m, t, n)
 1  d ← 1                          /* Prétraitement : calculs initiaux de la fonction de hachage */
 2  Pour i ← 1 Jusque m − 1
 3      Faire d ← d × 2
 4  hp ← 0
 5  Pour i ← 1 Jusque m
 6      Faire hp ← hp × 2 + p_i
 7  ht ← 0
 8  Pour i ← 1 Jusque m
 9      Faire ht ← ht × 2 + t_i
10
11  Si hp = ht cet TEST(p, m, t, 1)                    /* Recherche */
12      Alors AFFICHER(1)
13  h ← 2
14  TantQue h ≤ n − m + 1
15      Faire ht ← (ht − t_{h−1} × d) × 2 + t_{h+m−1}
16            Si hp = ht cet TEST(p, m, t, h)
17               Alors AFFICHER(h)
18            h ← h + 1
```

La complexité en temps de l'algorithme SM-KR reste en $O(nm)$ dans le pire des cas mais en pratique, le temps consacré est en $O(n+m)$ [4]. Il est facile de voir que le prétraitement est réalisé en temps $O(m)$. Pour le reste, le supplément de temps par rapport à l'algorithme SM-NAIF2 est limité à l'actualisation de la valeur ht lorsque la fenêtre coulisse, c'est une opération qui est réalisée en temps constant.

Exemple 2.4 *Considérons le texte* $t =$ `ATGTGTATTACCTATTAA` *et le motif* $p =$ `ATTA`. *Pour fixer les conventions, décidons que la lettre* `A` *a la "valeur" 1, la lettre* `C` *la "valeur" 2, la lettre* `G` *la "valeur" 3 et la lettre* `T` *la "valeur" 4*[3]

On a $hash(\texttt{ATTA}) = 1 \times 2^3 + 4 \times 2^2 + 4 \times 2^1 + 1 = 33$. *Les valeurs de la fonction de hachage pour chacune des positions de la fenêtre coulissante sont :*

Pos	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
Car	`A`	`T`	`G`	`T`	`G`	`T`	`A`	`T`	`T`	`A`	`C`	`C`	`T`	`A`	`T`	`T`	`A`	`A`
$hash$	34	55	50	53	46	48	***33***	52	42	24	***33***	38	48	***33***	51			
							**********							**********				

Dans cet exemple, trois fenêtres ont la bonne valeur de la fonction de hachage mais seulement deux d'entres-elle correpondent à une occurrence du motif. On voit, dans le cas de la fenêtre débutant en position 1*, que bien que le début de la fenêtre soit identique au motif, la valeur de la fonction de hachage permet de ne pas effectuer du tout de comparaison.*

2.6 Algorithme de Knuth-Morris-Pratt (1977)

L'algorithme de Knuth-Morris-Pratt est plus rapide car il effectue, dans certains cas, des décalages du motif de plus d'une position. De plus, il permet de ne pas relire les caractères

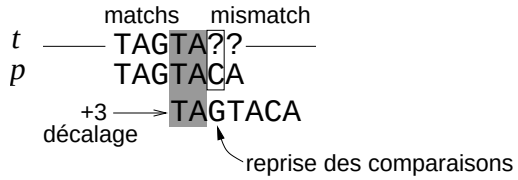
3. Ce choix a en fait peu d'importance, on pourraît par exemple prendre le code ASCII de chacune des lettres.

du texte qui ont déjà été comparés avec succès par l'algorithme. Grâce à cela, nous montrons que l'algorithme de Knuth-Morris-Pratt est linéaire dans le pire des cas.

2.6.1 Principe

Certains motifs réguliers (contenant des suites de caractères répétées) permettent d'effectuer des décalages de plus d'une position lorsqu'un mismatch se produit.

Exemple 2.5 *Soit $p =$ TAGTACA le motif. Imaginons, dans l'algorithme* SM-NAÏF2*, qu'un mismatch se produise en position 6 du motif. Indépendamment du texte, le motif pourrait être décalé directement de 3 positions en tenant compte des 5 matchs qui ont précédé le mismatch. En effet, ces 5 comparaisons ont été réussies car les 5 premiers caractères du texte concordent avec les 5 caractères correspondant du texte. Dès lors, pour qu'un décalage ait une chance de conduire à une position d'occurrence, les caractères du motif qui vont être amenés sous les caractères du texte impliqués dans le match doivent concorder :*



De plus, puisque ces caractères concordent, cela ne sert à rien de reprendre les comparaisons en début de motif, il faut reprendre les comparaisons là où elles s'étaient arrêtées dans le texte.

D'une manière générale, si un mismatch se produit entre le caractère courant du texte t_i et le caractère courant du motif p_j [4], cela signifie que :

— $t_{i-j+1..i-1} = p_{1..j-1}$: les $j-1$ premières comparaisons ont réussi entre le texte et le motif.

— $t_i \neq p_j$: mismatch à la position courante.

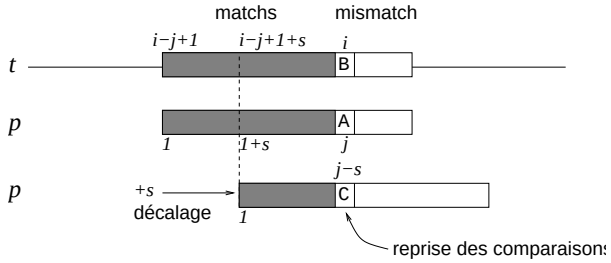
Nous dirons qu'un décalage de s positions vers la droite ($s > 0$) est *compatible* avec la configuration actuelle si :

A. $p_{j-s} \neq p_j$: le caractère du motif que le décalage amène sous le caractère courant du texte doit être distinct du caractère du motif qui a provoqué le mismatch.

B. $p_{1..j-s-1} = t_{i-j+1+s..i-1}$: les caractères du texte qui faisaient partie du match actuel continuent de concorder après le décalage.

Puisque ces caractères du texte sont identiques à ceux du motif qui se trouvent en face avant le décalage, cette condition B s'écrit également :

$$p_{1..j-s-1} = p_{1+s..j-1}$$



4. Noter le changement de notation : i est la position courante dans le texte et j est la position courante dans le motif.

Le décalage qui doit être appliqué est $s^* = min\{s|s \text{ compatible}\}$ car parmi toutes les possibilités, il faut prendre celle qui ne risque de manquer aucune occurrence, c'est-à-dire celle qui provoque le décalage minimal.

Les conditions A et B expriment donc qu'en cas de mismatch à la position courante j du motif, le décalage minimal s^* à effectuer ne dépend que du motif p et de la position j. Le décalage ne dépend absolument pas du texte. Dès lors, le motif va subir un *prétraitement* indépendant du texte qui va déterminer, pour chaque position possible de mismatch j dans le motif, la valeur du décalage à effectuer.

2.6.2 La table $next[]$

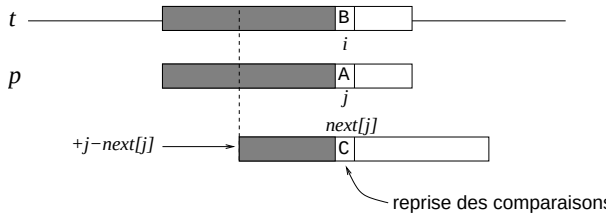
En cas de mismatch entre le caractère i du texte et le caractère j du motif, le décalage doit être effectué. Puisqu'on ne revient jamais en arrière dans le texte, seule la valeur de j doit être modifiée.

Définition 2.1 *Pour chaque position j de p, $next[j]$ indique la position, dans p, du caractère qui sera comparé avec le caractère courant de t après le mismatch. Le décalage appliqué est donc $j - next[j]$. La valeur $next[j]$ est la valeur maximale inférieure à j telle que le décalage $j - next[j]$ satisfasse les conditions A et B.*

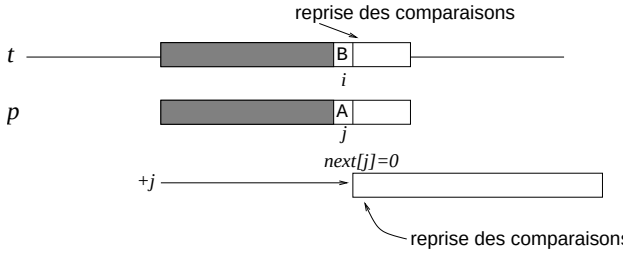
On définit également $next[m+1]$ comme la position du motif qui devra être comparée avec le caractère courant du texte après détection d'une occurrence du motif.

Une valeur $next[j] = 0$, avec $1 \leq j \leq m+1$ signifie qu'aucun décalage ne satisfait les conditions A et B.

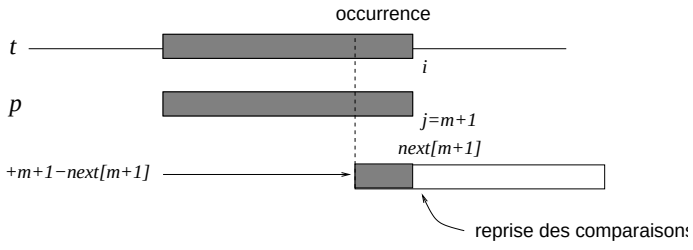
En cas de mismatch ou d'occurrence du motif, il suffit de remplacer la valeur j par $next[j]$.



Dans le cas où $next[j] = 0$, aucun décalage ne satisfait les conditions A et B, il faut alors abandonner la position courante du texte pour passer à la suivante et reprendre les comparaisons en début de motif.



En cas d'occurrence du motif, on reprend les comparaisons dans le motif à la position $next[m+1]$. Il est facile de voir que $next[m+1] = |Border(p)| + 1$:



Exemple 2.6 *Soit* $p =$ `TACTGTACTA`*. Les valeurs de la table* $next[]$ *sont :*

j	1	2	3	4	5	6	7	8	9	10	11
p	T	A	C	T	G	T	A	C	T	A	
$next[j]$	0	1	1	0	2	0	1	1	0	5	3

La valeur $next[1]$ sera toujours nulle car le seul décalage compatible dans le cas d'un mismatch en première position est celui qui recommence les comparaisons en début de motif.

2.6.3 Algorithme

Si la table $next[]$ est connue, l'algorithme de recherche de Knuth-Moriss-Pratt s'écrit simplement :

```
SM-KMP(p, m, t, n, next[])
1  i ← 1; j ← 1
2  TantQue i + m − j ≤ n
3     Faire TantQue j ≠ 0 cet p_j ≠ t_i
4           Faire j ← next[j]
5        i ← i + 1; j ← j + 1            /* Progression d'une position */
6        Si j > m
7           Alors AFFICHER(i − j + 1)
8                 j ← next[m + 1]
```

Chaque passage de la boucle extérieure permet d'avancer d'une position dans le texte. La condition d'arrêt permet de stopper dès que la position i est telle que la fenêtre actuelle "déborde" à droite du texte. Tout aura alors été testé.

La boucle intérieure effectue les décalages en cas de mismatch. Soit elle se termine car aucun décalage n'est compatible avec le caractère t_i (c'est-à-dire $j = 0$), soit parce qu'un décalage compatible a été trouvé ($p_j = t_i$).

Si la valeur de j dépasse m, cela signifie que tous les caractères du motif ont été parcourus, il y a donc une occurrence du motif en position $i - j + 1$ du texte.

Remarque 2.3 *Bien que la condition A interdise d'amener, sous la position* t_i*, le même caractère que celui qui a provoqué le mismatch, il se peut que plusieurs mismatchs se produisent successivement pour la même position du texte. En effet, le caractère différent amené peut lui aussi provoquer un mismatch comme le montre l'exemple suivant.*

Exemple 2.7 *Soit* $p =$ `abcabdabcabe` *le motif. Les valeurs de la table* $next[]$ *sont :*

j	1	2	3	4	5	6	7	8	9	10	11	12	13
p	a	b	c	a	b	d	a	b	c	a	b	e	
$next[j]$	0	1	1	0	1	3	0	1	1	0	1	6	0

Dans ce cas-ci, si un mismatch se produit en position 12 du motif, un autre mismatch peut se produire avec la nouvelle valeur de j *(6),* j *est alors remplacé par 3 et ainsi de suite (les mismatch sont représentés par des caractères en majuscule dans le motif) :*

```
t   adabcabdabcabfadabcabdabcabead
p     abcabdabcabE
      ->    abcabDabcabe
            -> abCabdabcabe
               -> aBcabdabcabe
                     .......
```

2.6.4 Complexité en temps de la recherche

Le théorème suivant montre que l'algorithme SM-KMP est linéaire dans le pire des cas.

Théorème 2.2 *La complexité en temps de* SM-KMP *est* $O(n)$ *lorsque la table* $next[]$ *est connue.*

Preuve

Chaque comparaison de l'algorithme conduit soit à effectuer un décalage (passage dans la boucle intérieure), soit avancer d'une position dans le texte (incrémentation de la variable i).

Soit n_i le nombre maximal d'incrémentations de la variable i et n_d le nombre maximal de décalages du motif. Clairement, la complexité en temps de SM-KMP est $O(n_i + n_d)$ puisque l'on compte le nombre de comparaisons.

La valeur initiale de i étant 1, n incrémentations au maximum peuvent être réalisées avant que la condition d'arrêt de la boucle extérieure soit verifiée. Dès lors, la valeur n_i vaut au maximum n.

La valeur n_d est également majorée par n car on peut décaler au maximum n fois le motif vers la droite (dans le pire des cas, on décale seulement d'une seule position à chaque mismatch).

L'algorithme SM-KMP fonctionne donc en temps $O(2n) = O(n)$. □

2.6.5 Prétraitement

Le calcul efficace de la table $next[]$ utilise les propriétés des bords des préfixes d'un mot (voir section 1.3).

Simplification du prétraitement : Moriss-Pratt

Le prétraitement de Morris-Pratt consiste à calculer une version simplifiée de la table $next[]$: on ne tient pas compte de la condition A, c'est-à-dire que l'on n'impose pas d'amener un caractère différent de celui qui a provoqué le mismatch sous la position courante du texte. Appelons cette table $next_{MP}[]$.

Imaginons donc le travail de la fonction SM-KMP utilisée avec la table $next_{MP}[]$ plutôt qu'avec la table $next[]$. Formellement, en cas de mismatch entre t_i et p_j, la valeur $next_{MP}[j]$ est le plus grand entier $k < j$ tel que $p_{1..k-1} = p_{j-k+1..j-1}$. Cette condition établit que $p_{1..k-1}$ est le bord maximal de $p_{1..j-1}$. Donc $next_{MP}[1] = 0$ et pour $1 < j \leq m+1$:

$$next_{MP}[j] = |Border(p_{1..j-1})| + 1$$

Le corollaire 1.4 nous propose un algorithme pour construire la table des bord maximaux des préfixes d'un mot. .

Soit $x \in \mathcal{A}^+$ un mot de longueur m. On définit sa table des bords maximaux $\beta[0..m]$ de la manière suivante :

— $\beta[0] = -1$

— $\beta[i] = |Border(x_{1..i})|$ pour $1 \leq i \leq m$.

La table des bords maximaux est calculée par l'algorithme BordsMaximaux. Il procède par le calcul des valeurs de $\beta[]$, de la gauche vers la droite, de telle sorte que les valeurs $\beta[]$ de préfixes plus courts soient utilisables (voir corollaire 1.4) lors du calcul du bord maximal d'un plus long préfixe.

```
BordsMaximaux(x, m, β[])
1  j ← 1; k ← −1
2  β[0] ← −1
3  TantQue j ≤ m
4      Faire TantQue k ≥ 0 cet x_j ≠ x_{k+1}
5          Faire k ← β[k]
6      k ← k + 1
7      β[j] ← k
```

8 $\quad j \leftarrow j + 1$

La boucle extérieure consiste à calculer la valeur β, de position en position. La variable j désigne la longueur du préfixe dont on calcule le bord maximal. La variable k contient la longueur du bord maximal de l'étape précédente. À chaque étape, le bord du préfixe de l'étape précédente est allongé d'une lettre. Si cette lettre correspond à la lettre courante de x, alors l'ancien bord maximal allongé d'une lettre est le bord maximal pour le préfixe courant. Dans le cas contraire, la boucle intérieure parcourt tous les bords du préfixe courant à la recherche de celui qui peut être allongé d'une lettre pour fournir le bord maximal du préfixe courant (voir corollaire 1.4).

On peut remarquer la similitude entre l'algorithme BORDSMAXIMAUX et l'algorithme de recherche SM-KMP. En fait, le principe est le même mais ici c'est le motif qui glisse le long de lui même. Par des arguments identiques à ceux de SM-KMP, la complexité en temps de BORDSMAXIMAUX est en $O(2m) = O(m)$.

Le prétraitement de Moriss-Pratt est donc réalisé simplement grâce à la fonction suivante dont la complexité en temps est $O(m)$ [5].

PREPRO-MP1$(p, m, next_{MP}[])$
1 BORDSMAXIMAUX$(p, m, \beta[])$
2 **Pour** $j \leftarrow 1$ **Jusque** $m + 1$
3 **Faire** $next_{MP}[j] = \beta[j-1] + 1$

ou

PREPRO-MP2$(p, m, next_{MP}[])$
1 $j \leftarrow 1; k' \leftarrow 0$
2 $next_{MP}[1] \leftarrow 0$
3 **TantQue** $j \leq m$
4 **Faire TantQue** $k' > 0$ **cet** $p_j \neq p_{k'}$
5 **Faire** $k' \leftarrow next_{MP}[k']$
6 $k' \leftarrow k' + 1; j \leftarrow j + 1$
7 $next_{MP}[j] \leftarrow k'$

Remarque 2.4 *L'algorithme de recherche utilisant la table $next_{MP}[]$ au lieu de la table $next[]$ fonctionne correctement mais risque de réaliser plusieurs mismatchs inutiles. Prenons par exemple la recherche de $p =$* `AACAACAAD`*. Supposons qu'à un moment déterminé, la fenêtre courante du texte t soit* `AACAABAAC` *:*

```
t   AACAABAAD
p   AACAACAAD
```

Il y aura un mismatch en position 6 du motif. Dans ce cas, $next_{MP}[6] = 3$ et donc le décalage va provoquer immédiatement un nouveau mismatch car c'est de nouveau un caractère `C` *qui est amené sous le* `B` *du texte :*

```
t   AACAABAAD
p      AACAACAAD
```

Intégration de la première condition : Knuth-Morris-Pratt

Le prétraitement de Knuth-Morris-Pratt consiste à intégrer la condition A au prétraitement pour éviter des mismatchs inutiles.

Il suffit pour cela d'adapter l'algorithme PREPRO-MP2 : lorsque $next[j]$ doit recevoir sa valeur (ligne 7), l'égalité entre p_j et $p_{k'}$ est testée. Si les deux caractères sont distincts, alors les conditions A et B sont satisfaites, $next[j]$ doit bien prendre comme valeur k'. Par contre, si les deux caractères sont identiques, la valeur k' ne respecte pas la condition A. Il suffit alors de donner à $next[j]$ la valeur $next[k']$. En effet :

— $next[k']$ est connu puisque k' est plus petit que j.
— Puisqu'un bord d'un bord est encore un bord (proposition 1.1), le fait que k' satisfasse la condition B implique que $next[k']$ la satisfasse aussi.

5. La version PREPRO-MP1 utilise la fonction BORDSMAXIMAUX tandis que la version PREPRO-MP2 intégre $next_{MP}[j] = \beta[j-1] + 1$. De plus, le changement de variable $k' = k + 1$ a également été réalisé pour alléger la notation.

— Puisque $next[k']$ a été calculé avant $next[j]$, il a subi le même traitement. Dès lors, la valeur $next[k']$ décrivant le plus petit décalage valide en cas de mismatch en k' respecte la condition A et donc $p_{k'} \neq p_{next[k']}$ (ou $next[k'] = 0$). Puisque $p_{k'} = p_j$ dans ce cas-ci, on a bien $p_j \neq p_{next[k']}$. La condition A est donc satisfaite par l'affectation $next[j] \leftarrow next[k']$.

L'algorithme de prétraitement de Knuth-Moriss-Pratt est le suivant.

```
PrePro-KMP(p, m, next[])
1  j ← 1; k' ← 0
2  next[1] ← 0
3  TantQue j ≤ m
4     Faire TantQue k' > 0 cet p_j ≠ p_k'
5              Faire k' ← next[k']
6           k' ← k' + 1; j ← j + 1
7           Si j > m cou p_j ≠ p_k'
8              Alors next[j] ← k'
9              Sinon  next[j] ← next[k']
```

Complexité en temps du prétraitement

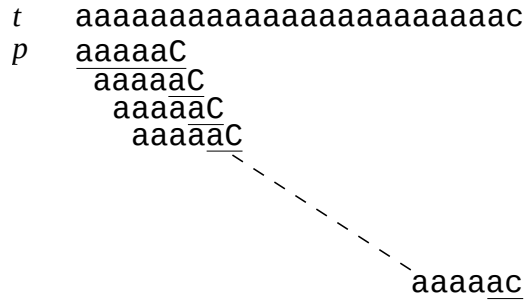
La fonction PrePro-KMP a une complexité en temps de $O(m)$. En effet, elle n'effectue aucune opération répétitive supplémentaire par rapport à l'algorithme BordsMaximaux.

Exemple frappant

Soit $p =$ `aaaaac` le motif et $t =$ `aaaaaaaaaaaaaaaaaaaaaaac` le texte ($|t| = 24$).
Les valeurs de la table $next[]$ sont :

j	1	2	3	4	5	6	7
p	a	a	a	a	a	c	
$next[j]$	0	0	0	0	0	5	1

La recherche de toutes les occurrences de p dans t produit l'enchaînement suivant :



Les comparaisons sont soulignées, les mismatchs sont représentés par des lettres en majuscule. Bien que le décalage ne soit que de 1 à chaque étape, le gain de temps provient du fait que les comparaisons réussies ne sont pas recommencées. Le nombre de comparaisons effectuées ici est de 42. Il aurait fallu 114 comparaisons dans le cas de l'algorithme SM-Naïf2 !

2.7 Algorithme de Boyer-Moore (1977)

Nous ne développons pas cet algorithme ici.

2.7.1 Principe

Comme pour l'algorithme de Knuth-Moriss-Pratt, la fenêtre coulisse de la gauche vers la droite mais l'originalité est ici de réaliser les **comparaisons de la droite vers la gauche** à l'intérieur de la fenêtre. En cas de mismatch, le motif peut-être décalé de plus d'une position vers la droite en fonction de critères semblables à ceux de l'algorithme de Knuth-Moriss-Pratt : il faut mettre en correspondance des caractères que l'on sait concorder.

Le décalage à appliquer en cas de mismatch ne dépend que du motif, de la position de mismatch et du caractère qui a provoqué le mismatch. Dès lors, le motif est prétraité avant la recherche pour connaître les décalages à effectuer en cas de mismatch. Le temps de prétraitement est $O(m+q)$ où $q = \#\mathcal{A}$ est la taille de l'alphabet.

Dans le pire des cas, la complexité en temps de la recherche est $O(nm)$. L'intérêt de l'algorithme de Boyer-Moore provient surtout de sa complexité moyenne : la recherche est **sous linéaire en moyenne**, c'est-à-dire que tous les caractères du texte ne sont pas forcément lus. En effet, lorsqu'un décalage du motif est effectué de la gauche vers la droite, les caractères du texte, situés à gauche dans la fenêtre, qui n'avaient pas encore été lus et qui sont maintenant "sortis" de la fenêtre ne seront plus jamais considérés.

Le prétraitement est très complexe d'un point de vue technique, les preuves de complexité temporelle également. Une modification de l'algorithme a été proposée qui permet, au moyen d'un prétraitement plus compliqué, d'obtenir un temps linéaire dans tous les cas [1]. R. Cole a prouvé qu'avec cet algorithme, le nombre de comparaisons est toujours borné par $3n$ [2].

Bibliographie

[1] D. Beauquier, J. Berstel, Ph. Chrétienne, *Éléments d'algorithmique*, Masson, 1992.

[2] R. Cole, Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.*, 23 :1075–91.

[3] M. Crochemore, C. Hancart, T. Lecroq, *Algorithmique du texte*, Vuibert, 2001

[4] M. Crochemore, T. Lecroq, *Pattern Matching and Text Compression Algorithms*, in *The Computer Science and Engineering Handbook*, Allen B. Tucker ed, CRC Press 2003.

[5] *Génet, Université de Tours, 2000-2002*, `http://www.univ-tours.fr/genet/gena.htm`

[6] *MolGraph : the Picture Collection*, `http://wwwchem.leidenuniv.nl/metprot/armand/pict.html`, *Leiden University, The Nederlands, 2000*.