3.6 Algorithmes de recherche dans les chaînes de caractères

Quand on utilise des chaînes de caractères, on a souvent besoin d'examiner une chaîne pour y rechercher une sous-chaîne donnée. Une telle opération de recherche existe dans tous les logiciels de traitement de textes, ainsi d'ailleurs que dans toutes les réalisations du type de données abstrait *chaîne de caractères*. Lorsqu'on manipule de longs textes, on doit réaliser cette opération aussi efficacement que possible. Nous nous concentrerons ici sur quatre algorithmes classiques de recherche de patron[1] (en anglais *pattern-matching*), qui examinent une chaîne pour y trouver une sous-chaîne donnée.

Recherche simple

La réalisation de l'opération `find`, pour prendre une des opérations de recherche associées au type `string`, permet d'illustrer l'algorithme de recherche de patron le plus simple. La figure 3.6 illustre cet algorithme pour la recherche de la sous-chaîne "abc" dans la chaîne "ababc"; le caractère "|" marque les positions dans le patron et dans la chaîne où la comparaison est effectuée. On commence au début de la chaîne et de la sous-chaîne (ou patron) et on compare les premiers caractères. Ils correspondent; on compare alors les deuxièmes caractères, ils correspondent aussi. On compare donc les troisièmes caractères; comme ils ne correspondent pas (indiqué par un astérisque), on revient au deuxième caractère de la chaîne et au premier caractère du patron et on répète le processus.

```
|      |       |      |       |       |       |
ababc ababc   ababc   ababc   ababc   ababc  ababc       Chaîne de caractères
abc    abc     abc      abc     abc     abc    abc        Patron
|1     |2       |3*    |4*     |5       |6      |7       Comparaisons
```

Figure 3.6 Recherche de patron

Cet algorithme est simple et facile à programmer. La fonction `RechercheSimple` ci-dessous illustre cet algorithme de façon plus détaillée :

```
int RechercheSimple(const string& Patron, const string& Texte)
{
  int carac, pat, longueurPatron, longueurTexte;
  longueurPatron = Patron.size();
  longueurTexte = Texte.size();
  pat = 0;
  if(longueurPatron <= longueurTexte){
    carac = 0;
    do{
      if(Patron[pat] == Texte[carac]){
        pat++;
        carac++;
      }
      else{
        carac = carac - pat + 1;  // avance dans Texte
        pat = 0;
      } //if
    } while(pat < longueurPatron && carac < longueurTexte);
  } //if;
  if((pat >= longueurPatron))   // trouvé
    return carac - longueurPatron;
  else
    return -1;
} //RechercheSimple;
```

[1] Sous-produit d'une recherche d'emploi?

Dans la boucle `do-while`, les caractères de la chaîne et du patron sont comparés un à un; chaque fois qu'on note une différence des caractères, on repart au début du patron après avoir reculé dans la chaîne. Dans le pire des cas, c'est-à-dire celui où le patron ne se trouve pas dans la chaîne, chacun des *p* caractères du patron sera comparé aux *t* caractères de la chaîne texte (par exemple si nous cherchons "*hipi*" dans "*hipahipbhipchipdhipehipfhipg*"). La complexité temporelle de l'algorithme est donc *O(pt)*.

Algorithme de Knuth-Morris-Pratt

En examinant la recherche de patron de la figure 3.6, on se rend facilement compte que certaines comparaisons sont inutiles. Par exemple, la comparaison de l'étape 4 est inutile, car le second caractère de la chaîne a déjà été examiné et peut être sauté. L'idée de base de l'algorithme de Knuth-Morris-Pratt[2] est de tirer avantage de l'information obtenue avant qu'une différence ne soit trouvée. Si une différence apparaît au n^{ème} caractère du patron, les (n-1) premiers caractères correspondaient. Sauter tous ces caractères ne fonctionnera pas, car le patron pourrait correspondre à des parties de lui-même au point où la différence se produit. Par exemple, si nous cherchons "grigou" dans "grisettegrignotanteetgrigou", la première différence se produit au quatrième caractère, nous avançons dans la chaîne et reprenons le patron au début. La seconde différence après quelques correspondances se produit au treizième caractère de la chaîne. On reste au même endroit de la chaîne et on repart au second caractère du patron, un peu comme si on avait reculé dans la chaîne et repris au début du patron. Le meilleur « recul » ne dépend que du patron et du fait qu'il peut se correspondre, constatation qui peut être faite au préalable et conservée dans une table des reculs; la figure 3.7 montre ce qu'il faut faire si un caractère du patron ne correspond pas au caractère du texte et aide à comprendre le recul dans le patron.

```
 Position  Correspondance Recul[j]          Commentaires
différence
    0          aucune       -1    pas g, avancer dans chaîne et patron au début
    1          g             0    pas r, peut être g, patron: aller au début, chaîne: rester
    2          gr            0    pas i, peut être g, patron: aller au début, chaîne: rester
    3          gri          -1    pas g, avancer dans chaîne et patron au début
    4          grig          1    pas o, peut être r, patron: aller à 1, chaîne: rester
    5          grigo         0    pas u, peut être g, patron: aller au début, chaîne: rester
```

Figure 3.7 Table des reculs

De façon un peu plus générale, lorsqu'on calcule le recul (ou décalage) à effectuer on peut raisonnablement penser qu'un préfixe *pré* du patron correspond à un suffixe de la portion du patron ayant correspondu au texte *cor* (figure 3.8). Afin d'éviter une non correspondance immédiate, il faut que le caractère qui suit immédiatement *pré* soit différent du caractère du texte ayant provoqué la non correspondance après *cor*. On appelle le plus long de ces préfixes la bordure de *cor*. On donne à TableRecul[i] la valeur de la longueur de la bordure de Patron[0..i-1] suivie d'un caractère différent de Patron[i] ou –1 si une telle bordure n'existe pas. On reprend alors la comparaison entre Patron[TableRecul[i]] et Texte[i+j] sans manquer de correspondance de Patron dans Texte et en évitant de reculer dans le texte.

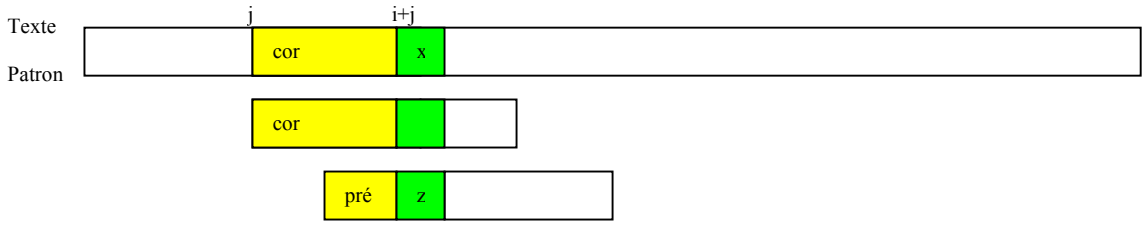

Figure 3.8 Décalage de l'algorithme de Knuth-Morris-Pratt

[2] Knuth, D. E., Morris, J. H., Pratt, V. R., *Fast pattern matching in strings*, SIAM J. Comp., vol. 6 n^{o} 2, juin 1977.

La fonction ci-dessous effectue la recherche de patron en avançant de la façon normale tant que les caractères correspondent; autrement elle recule dans le patron de la valeur trouvée dans une table des reculs. La figure 3.9 illustre le processus de décalage aux endroits où il y a certaines correspondances. Un astérisque indique la fin d'une correspondance.

Après la première différence au quatrième caractère, on avance de caractère en caractère à cause des différences jusqu'à une correspondance partielle qui s'arrête à une différence sur le cinquième caractère du patron; on recule au deuxième caractère du patron et on reprend dans la chaîne sur le caractère déjà examiné. La différence qui se produit nous fait à nouveau progresser de caractère en caractère jusqu'à la correspondance totale de la fin du patron.

```
grisettegrignotanteetgrigou
grigou.....................
...*grigou.................
....*grigou................
.... *grigou...............
......*grigou..............
.......*grig*u.............
...........g*igou..........
............*grigou........
.............*grigou......
..............*grigou.....
...............*grigou....
................*grigou...
.................*grigou..
..................*grigou.
...................*grigou
```

Figure 3.9 Recherche de “grigou”

```
void CalculerRecul(const string& Patron, int TableRecul[])
{ // Calculer la table des sauts pour les caractères du patron.
  int longueurPatron, pat1 = 0, pat2 = -1;
  longueurPatron = Patron.size();
  TableRecul[0] = -1;
  while(pat1 < longueurPatron){
    while(pat2 > -1 && Patron[pat1] != Patron[pat2])
      pat2 = TableRecul[pat2];
    pat1++;
    pat2++;
    if(Patron[pat1] == Patron[pat2])
      TableRecul[pat1] = TableRecul[pat2];
    else
      TableRecul[pat1] = pat2;
  }
} //CalculerRecul;

int KnuthMorrisPratt(const string& Patron, const string& Texte)
// Recherche Patron dans Texte.  Retourne indice du patron dans le texte
// sinon retourne -1.
{
  int carac, pat, longueurPatron, longueurTexte;
  int TableRecul[PATRONMAX];
  CalculerRecul(Patron, TableRecul);
  longueurTexte = Texte.size();
  longueurPatron = Patron.size();
  carac = 0;
  pat = 0;
```

```
  while(carac < longueurTexte){
    while(pat > -1 && Patron[pat] != Texte[carac])
      pat = TableRecul[pat];
    pat++;
    carac++;
    if(pat >= longueurPatron)
      return carac-pat;
  }
  return -1;
} //KnuthMorrisPratt
```

La procédure `CalculerRecul` est très semblable à la procédure `KnuthMorrisPratt`, car en essayant de faire correspondre le patron à lui-même, elle utilise la même méthode. La procédure `KnuthMorrisPratt` a une complexité de *O(t)*, où *t* est la longueur du texte, ce qui n'inclut pas la création de la table de recul qui est *O(p)*. La complexité totale est *O(p+t)*, ce qui est mieux que *O(pt)*. Cependant, ces procédures ne seront pas nécessairement bien plus rapides que la méthode de recherche simple lorsqu'elles seront appliquées à des textes réels, car ces derniers ne mettent pas souvent en jeu la recherche d'un patron très répétitif dans un texte très répétitif. Nous développerons davantage cette comparaison dans l'étude de cas qui suit.

Algorithme de Boyer-Moore

Il est encore possible d'améliorer la recherche en balayant le patron de droite à gauche et en décidant de la marche à suivre lorsqu'on rencontre la première différence, comme le fait l'algorithme de Boyer-Moore[3]. Supposez que nous prenions comme exemple la recherche de "gigogne" dans la chaîne "gigantesque gigolo gigotant dans le lit gigogne". Dans la forme simplifiée de cet algorithme, nous commençons par comparer le "e" du patron à la lettre "e" de la chaîne. Il y a correspondance; on compare ensuite le "n" et le "t" et on note une différence. Puisqu'il n'y a pas de "t" dans le patron nous pouvons avancer dans la chaîne de la longueur du patron. La prochaine comparaison est alors entre "e" et "i" et, comme "i" apparaît dans le patron, nous avançons le patron de cinq positions pour que son "i" corresponde au "i" de la chaîne, ce qui nous conduit à comparer "e" et " ". Comme il n'y a pas d'espace dans le patron, nous avançons alors encore de la longueur du patron pour comparer "e" et "n", puis nous avançons d'une position et comparons "e" et "t", avant d'avancer du patron entier. Nous comparons alors "e" et "l". Comme il n'y a pas de "l" dans le patron, nous avançons de la longueur du patron et nous comparons à nouveau "e" et "g". Comme il y a un "g" dans le patron nous avançons de deux positions et comparons encore "e" et "g", avançons encore de deux positions et nous trouvons la correspondance, comme le montre la figure 3.10.

```
gigantesque gigolo gigotant dans le lit gigogne
gigog*e.....................................
.......gigogn*..............................
...........gigogn*..........................
...................gigogn*...................
...................gigogn*...................
.........................gigogn*.............
.................................gigogn*.....
.....................................gigogn*....
.......................................gigogn*..
.........................................gigogne
```

Figure 3.10 Correspondance du patron à partir de la droite

Le décalage sur mauvais caractère fait aligner Texte[i+j] avec son occurrence la plus à droite dans le patron, comme le montre la figure 3.11.

[3] Boyer, R. S., Moore, J. S., *A fast searching algorithm*, CACM, vol. 20, n° 10, octobre 1977.

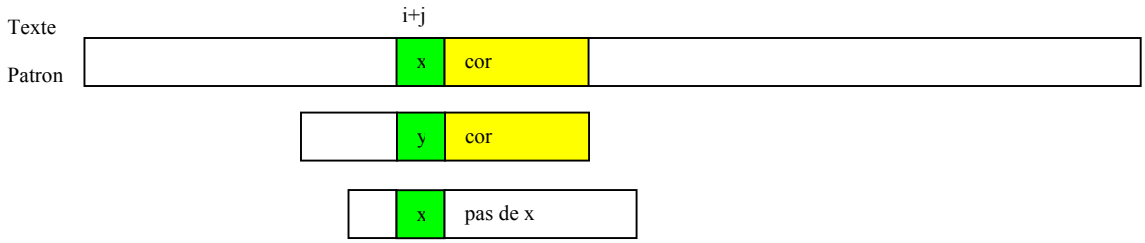

Figure 3.11 Décalage sur mauvais caractère réapparaissant

Si Texte[i+j] n'apparaît pas dans le patron, le patron doit être décalé au delà de Texte[i+j] comme la montre la figure 3.12. Les décalages résultant de cette méthode sont rangés dans une table des sauts de telle façon que TableSauts[car] prennent la valeur minimum de i telle que `Patron[longueurPatron-i-1] = car` ou `longueurPatron` si car n'apparaît pas dans Patron.

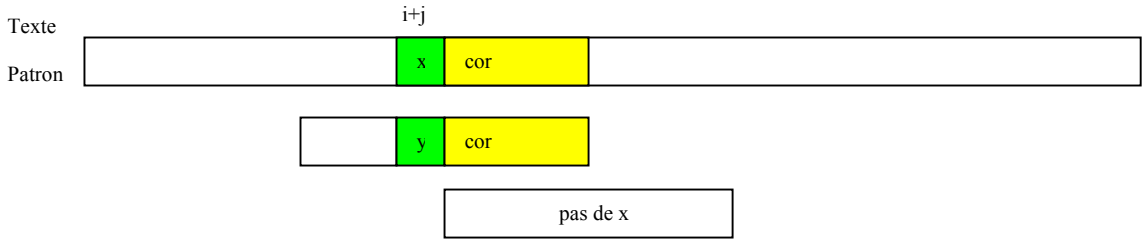

Figure 3.12 Décalage sur mauvais caractère non présent

Cette méthode pourrait donner des reculs du patron ; ceci ne se produira cependant pas si on utilise aussi la seconde méthode attachée à l'algorithme de Boyer-Moore. Cette seconde méthode recherche ce qu'on appelle des suffixes dans le patron. Si une différence se produit entre les caractères `Patron[i]` et `Texte[j+i]` on essaye d'aligner le « bon » suffixe `Texte[j+i+1..j+lP-1]` avec son occurrence la plus la plus droite dans le patron qui sera précédée d'un caractère différent de `Patron[i]` (voir figure 3.13).

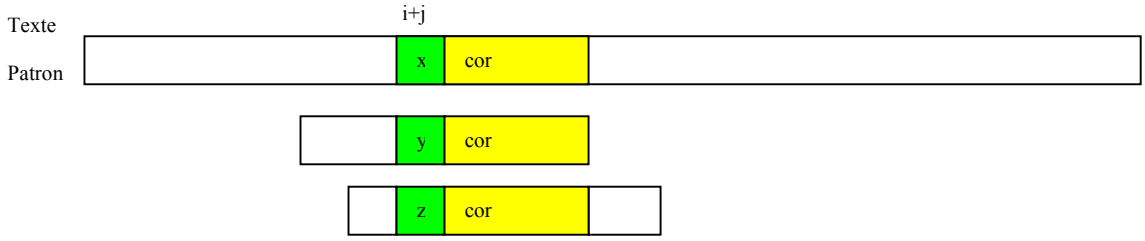

Figure 3.13 Décalage du bon suffixe présent plusieurs fois

Évidemment, si le bon suffixe ne réapparaît pas, on aligne le plus long suffixe possible de `Texte [j+i+1..j+lP-1]` avec un préfixe correspondant du patron (voir figure 3.14).

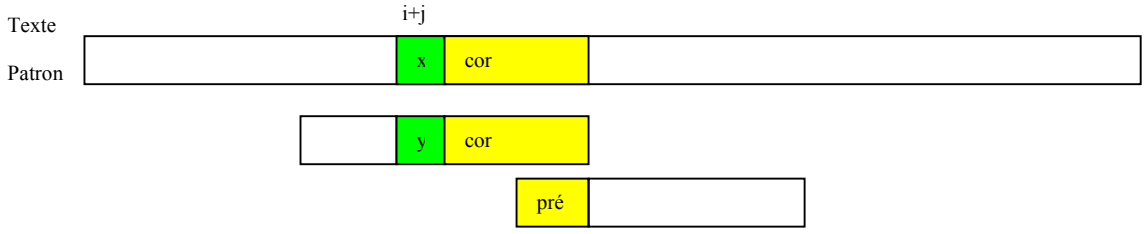

Figure 3.14 Décalage si le bon suffixe n'est pas présent plusieurs fois

Les décalages calculés par cette seconde méthode sont rangés dans une table des suffixes. Pour la calculer on remplit d'abord les éléments d'un tableau `Suffixes[k]` avec la valeur

maximum de i telle que `Patron[k-i-1..i] = Patron[longueurPatron-i..longueurPatron-1]`. La table des suffixes est alors calculée en y plaçant des valeurs telles que `TableSuffixes[i+1]` (i variant de 0 à longueurPatron-1) reçoive la valeur minimum positive de k telle que pour tout j compris entre i et longueurPatron k ≥ j ou `Patron[j-k] = Patron[j]` et si k < i `Patron[i-k] ≠ Patron[i]`.

Nous avons comparé 17 caractères du texte, y compris 7 pour déterminer la correspondance finale. L'algorithme correspondant est réalisé par la fonction `BoyerMoore` ci-dessous. Elle utilise le vecteur `TableSauts` défini plus haut. Dans cette table, tous les caractères qui n'apparaissent pas dans le patron donnent un saut de la longueur du patron, tandis que les caractères qui apparaissent dans le patron donnent des sauts allant de zéro à la longueur du patron moins 1. Par exemple, pour notre patron “gigogne” `TableSauts` donne zéro pour la lettre “e”, 1 pour la lettre “n”, 2 pour la lettre “g”, 3 pour la lettre “o”, 5 pour la lettre “i” (il n'y a pas de 4 ou de 5, puisque la lettre “g” est répétée dans le patron). La procédure `CalculerSauts` crée cette table des sauts. Le vecteur `TableSuffixes` est calculé par la procédure `CalculerSuffixes` selon la méthode exposée plus haut, qui est moins intuitive que la précédente.

```
void CalculerSauts(const string& Patron, int TableSauts[])
{ // Calculer les décalages dûs aux non correspondances
  int longueurPatron = Patron.size();
  for(int i = 0; i < TAILLEALPHA; ++i)
    TableSauts[i] = longueurPatron;
  for(int i = 0; i < longueurPatron - 1; ++i)
    TableSauts[Patron[i]] = longueurPatron - i - 1;
}//CalculerSauts

void CalculerSuffixes(const string& Patron, int Suffixes[])
{ // Établir la tables des suffixes du patron
  int longueurPatron = Patron.size();
  int iprec, index;
  Suffixes[longueurPatron - 1] = longueurPatron;
  index = longueurPatron - 1;
  for(int i = longueurPatron-2; i >= 0; i--){
    if(i > index && Suffixes[i+longueurPatron-1-iprec] < i-index)
      Suffixes[i] = Suffixes[i+longueurPatron-1-iprec];
    else{
      if(i < index)
        index = i;
      iprec = i;
      while(index >= 0 && Patron[index] == Patron[index + longueurPatron-1-iprec])
        index--;
      Suffixes[i] = iprec - index;
    }
  }
}//CalculerSuffixes

void CalculerTableSuffixes(const string& Patron, int TableSuffixes[])
{ // Calculer les valeurs de la table des suffixes
  int Suffixes[PATRONMAX];
  int longueurPatron = Patron.size();
  CalculerSuffixes(Patron, Suffixes);
  for(int i = 0; i < longueurPatron; i++)
    TableSuffixes[i] = longueurPatron;
  int j = 0;
  for(int i = longueurPatron-1; i >= -1; i--)
    if(i == -1 || Suffixes[i] == i+1)
      for(; j < longueurPatron-1-i; j++)
        if(TableSuffixes[j] == longueurPatron)
          TableSuffixes[j] = longueurPatron-1-i;
```

```
  for(int i = 0; i <= longueurPatron - 2; i++)
    TableSuffixes[longueurPatron-1-Suffixes[i]] = longueurPatron-1-i;
}//CalculerTableSuffixes

int BoyerMoore(const string& Patron, const string& Texte)
{ // Appliquer l'algorithme de recherche de Boyer-Moore
  int i, tableSuffixes[PATRONMAX], tableSauts[TAILLEALPHA];
  int longueurPatron = Patron.size();
  int longueurTexte = Texte.size();
  CalculerSauts(Patron, tableSauts);
  CalculerTableSuffixes(Patron, tableSuffixes);
  int j = 0;
  while(j <= longueurTexte-longueurPatron){
    for(i = longueurPatron-1;
        i >= 0 && Patron[i] == Texte[i+j];
        i--);  // reculer dans chaîne
    if(i < 0){ // correspondance totale
      return j; // ou j += tableSuffixes[0];  pour poursuivre recherche
    }
    else{       // caractères différents
      unsigned char carac = Texte[i+j];
      j += max(tableSuffixes[i], tableSauts[carac]-longueurPatron+1+i);
    }
  }
  return -1;
}//BoyerMoore
```

La fonction `BoyerMoore` essaye de faire correspondre le patron de droite à gauche. Lorsqu'une différence se produit, la fonction avance dans la chaîne en choisissant la plus grande valeur de saut à partir des valeurs de sauts de la table des sauts et des valeurs de décalage de la table des suffixes. Cet algorithme de Boyer-Moore est l'algorithme de recherche de patron le plus rapide des algorithmes connus: avec un long alphabet et des patrons courts, la complexité temporelle de l'algorithme est approximativement *O(t/p)*.

Algorithme de Rabin-Karp

L'algorithme de Rabin-Karp[4] utilise une fonction de transformation ou de hachage pour calculer une valeur numérique pour le patron. L'algorithme calcule une valeur numérique de hachage (voir le chapitre 11) pour chaque tranche de *p* caractères possible de la chaîne, et vérifie si elle est égale à la valeur de hachage du patron. Cette méthode n'est pas très différente de celle de la recherche simple, sauf que le calcul de la valeur de hachage pour la position *i* est basé sur la valeur de hachage de la position *i-1*.

On considère les *p* caractères comme les parties d'un entier, chaque caractère étant interprété comme un chiffre en base *b*, où *b* est le nombre de caractères différents possibles. La valeur correspondant aux *p* caractères entre les indices *k* et *k+p-1* est exprimée par:

$$n = t_k b^{p-1} + t_{k+1} b^{p-2} + \ldots + t_{k+p-1}$$

Si on décale d'une position vers la droite dans le texte, la nouvelle valeur est:

$$(n - t_k b^{p-1})b + t_{k+p}$$

Nous prenons une fonction de hachage h(i) = i % N, où N est un grand nombre premier. Sachant que prendre le reste de la division par N après chaque opération arithmétique donne le même résultat que de prendre le reste de la division par N, une fois toutes les

[4] Karp, R. M., Rabin, M. O., *Efficient randomized pattern-matching algorithms*, TR-31-81, Aiken Comp. Lab., Harvard University, 1981.

opérations arithmétiques effectuées, nous pouvons programmer la fonction de recherche de la façon suivante :

```
int RabinKarp(const string& Patron, const string& Texte)
// Recherche de Patron dans Texte. Retourne l'indice du patron dans le texte
// ou zéro si pas trouvé.
{
  const int PREMIER = 5000011;  // gros nombre premier
  const int BASE = 256;
  int hachePatron, hacheTexte, puissance;
  int longueurTexte, longueurPatron, index;
  longueurPatron = Patron.size();
  longueurTexte = Texte.size();
  puissance = 1;
  for(int compteur = 1; compteur < longueurPatron; compteur++)
    // calculer BASE à la puissance (longueurPatron-1)
    puissance = (puissance * BASE) % PREMIER;
  hachePatron = 0;
  for(int indx = 0; indx < longueurPatron; indx++)
    // calculer nombre de hachage du patron
    hachePatron = (hachePatron * BASE + Patron[indx]) % PREMIER;
  hacheTexte = 0;
  for(int indx = 0; indx < longueurPatron; indx++)
    // calculer nombre de hachage du texte
    hacheTexte = (hacheTexte * BASE + Texte[indx]) % PREMIER;
  index = 0;
  while(hachePatron != hacheTexte &&
        index < longueurTexte - longueurPatron + 1){
    // calculer nouveau nombre de hachage de la tranche de texte
    hacheTexte = (hacheTexte + BASE * PREMIER -
                  Texte[index] * puissance) % PREMIER;
    hacheTexte = (hacheTexte * BASE +
                  Texte[index + longueurPatron]) % PREMIER;
    index++;
  } //while;
  if(index < longueurTexte - longueurPatron + 1)
    return index;
  else
    return -1;
} //RabinKarp;
```

Comme le code ASCII étendu normalisé comprend 256 codes, on a pris 256 comme base et le premier nombre premier supérieur à cinq millions[5] pour la constante PREMIER. Lorsque nous choisissons BASE et PREMIER, nous devons nous assurer que le calcul de (BASE+1) × PREMIER ne cause pas de débordement. Cette fonction calcule d'abord la valeur de hachage du patron ainsi que b^{p-1} , puis la valeur de hachage des *p* premiers caractères du texte. À partir de là, la fonction calcule de façon répétée la valeur de hachage de la prochaine section du texte, en utilisant la méthode présentée ci-dessus, jusqu'à ce qu'une valeur égale à celle du patron soit trouvée. Lorsqu'on trouve une correspondance, on doit encore effectuer une comparaison directe du patron et du texte, puisqu'il est possible que des chaînes différentes aient la même valeur de hachage. Cependant, une grande valeur pour PREMIER rend ce fait extrêmement peu probable. Dans la première étape du calcul, on ajoute BASE × PREMIER pour que la soustraction produise un résultat positif qui ne fausse pas le résultat de l'opération %. Cet algorithme a une complexité de $O(p+t)$, comme l'algorithme de Knuth-Morris-Pratt.

[5] Calculé par notre collègue mathématicien Gilbert Labelle

3.7 Étude de cas : performance des algorithmes de recherche de patrons

Nous venons de présenter quatre algorithmes classiques de recherche de patron dans une chaîne: recherche simple, Knuth-Morris-Pratt, Boyer-Moore et Rabin-Karp. Le choix d'algorithme dépendra probablement de l'application. Il est vrai que le pire des cas pour la complexité de la recherche simple est $O(n^2)$, si on cherche le patron xxxxxxxxxy dans la chaîne xxxxxxxxxxxxxxxxxxxx, mais on peut se demander combien de fois un tel cas se produit dans la réalité. Si la recherche met en jeu des patrons de bits faits de zéros et de uns, où les zéros et les uns ont tendance à être groupés, alors la recherche simple peut en effet connaître son pire comportement. Cependant, étant donné une chaîne de caractères qui apparaît avec la même fréquence que les caractères en français, le scénario du pire des cas aurait peu de chance de se produire. Dans cette étude de cas, nous étudierons la performance des quatre algorithmes de recherche de patron en utilisant des chaînes de caractères alphabétiques.

Comme la recherche de patron est un facteur critique dans un produit logiciel, il est important de coder ces divers algorithmes et d'en mesurer la performance. Notre objectif est un peu moins ambitieux : nous simulerons la recherche de patrons dans un texte où la fréquence des lettres est la même qu'en français.

Les fréquences des lettres dans un groupe de textes français typiques sont données par la figure 3.15[6].

a	0.0747	j	0.0060	s	0.0850
b	0.0087	k	0.0001	t	0.0708
c	0.0316	l	0.0569	u	0.0683
d	0.0367	m	0.0304	v	0.0152
e	0.1766	n	0.0724	w	0.0002
f	0.0111	o	0.0540	x	0.0037
g	0.0077	p	0.0276	y	0.0026
h	0.0081	q	0.0134	z	0.0013
i	0.0738	r	0.0633		

Figure 3.15 Fréquences des lettres en français

Ces fréquences ont été calculées à partir d'un grand ensemble de prose diverse. Comme nous l'avons déjà fait remarquer, les fréquences peuvent ne pas être les mêmes pour d'autres sortes de textes, comme des textes techniques ou même, bien pire, des textes de programmes. Les fréquences des textes peuvent aussi différer d'un auteur à l'autre, puisque, dans une certaine mesure, les fréquences d'utilisation des lettres peuvent caractériser le style d'un auteur. Dans notre étude de cas, nous utiliserons les fréquences de la figure 3.15. Le texte de nos recherches sera fait de suites de caractères, que nous appellerons des "mots", séparés par des espaces. Il n'y aura pas de ponctuation ou de caractères autres que des lettres, en dehors des 26 lettres minuscules et de l'espace.

La distribution de la longueur des mots dans le texte devrait être:

longueurs de 2 à 7 : 12.5 % pour chacune
longueurs de 8 à 10 : 5.0 % pour chacune

Ces distributions sont arbitraires et ne correspondent pas à des données vraies pour la longueur des mots en français.

Le programme engendrera un texte représenté par une chaîne de caractères de longueur 20 000, basée sur les données ci-dessus, et utilisera les quatre algorithmes de recherche présentés plus haut, pour rechercher des patrons dont la longueur variera entre 4 et 15 caractères. Les patrons ne comprendront pas d'espaces. Comme nous voulons mesurer le

[6] Muller, A., *Le décryptement*, Que sais-je?, PUF, 1983

pire des cas, nous mesurons le temps pour des recherches de patrons qui *ne sont pas présents dans la chaîne*. Le programme affichera une table des temps résultants basés sur les moyennes pour cent patrons différents de chaque longueur.

Après avoir effectué ces mesures, nous essayerons d'en tirer des conclusions pour chacun des quatre algorithmes utilisés.

Notons tout de suite que ni le texte ni les patrons ne seront vraiment du texte français. La fréquence des lettres ne peut, à elle seule, garantir la génération de français; les combinaisons de lettres sont en effet plus importantes que les fréquences individuelles dans une vraie langue. À la fin de cette étude de cas, nous indiquerons quelques façons d'engendrer du texte plus réaliste.

La solution retenue est assez simple:

- Engendrer une chaîne.
- Appliquer chaque algorithme systématiquement à 100 patrons de même longueur, la longueur variant de 4 à 15.
- Afficher une table des résultats moyens.

Pour accomplir ces tâches, nous aurons besoin d'une ensemble de procédures pour:

- engendrer des entiers aléatoires dans un intervalle donné;
- engendrer des lettres aléatoires avec la distribution de fréquences désirée;
- engendrer des suites de lettres aléatoires (“mots”) pouvant former un texte ou un patron;
- chronométrer chaque algorithme de recherche.

Nous avons utilisé des fonctions de génération de nombres aléatoires au début du chapitre, et nous les réutiliserons. La génération de lettres aléatoires avec une fréquence donnée sera faite en engendrant un entier aléatoire entre 1 et 1000 et en utilisant cette valeur pour choisir le caractère approprié. Par exemple, selon la figure 3.15, il y aurait 74 chances sur 1000 pour un *a*, 8 chances pour 1000 pour un *b*, 31 chances sur 1000 pour un *c*, etc. Ainsi les entiers allant de 1 à 74 correspondraient à un *a*, de 75 à 82 à un *b*, de 83 à 113 à un *c*, et ainsi de suite. Une instruction `if` pourra trouver le cas correspondant et retourner la bonne lettre.

Pour former des mots, nous engendrons une longueur selon la distribution choisie, puis nous produirons ce nombre de caractères. La génération d'une chaîne de caractères mettra en jeu la génération de mots séparés par des espaces jusqu'à ce qu'on atteigne la longueur voulue. La génération des patrons de recherche se fera en engendrant des lettres jusqu'au nombre de lettres correspondant à la longueur du patron.

Pour chronométrer chaque recherche, nous utiliserons l’opération `clock()` offerte par la bibliothèque `ctime` qui nous permettra de relever l'heure avant et après une recherche, puis de déterminer la différence en millisecondes entre ces deux heures. Bien que nous voulions être sûrs que le temps mesuré a été effectivement consacré à la recherche, les systèmes utilisés peuvent ne pas nous le permettre. Pour en être relativement sûrs, nous pouvons au moins nous assurer qu'en dehors du système d'exploitation aucune autre application n'est active au moment de la mesure.

Nous rangerons ces mesures dans un tableau à trois dimensions, indexé par le type de recherche, la longueur du patron et le nombre d'essais.

Le pseudocode ci-dessous affichera notre table des moyennes.

```
Afficher les moyennes résultats
    Afficher les en-têtes
    Pour chaque longueur de patron boucler
        Pour le nombre d'essais voulu boucler
            Choisir Patron
            Pour chaque type de recherche boucler
                Mettre le temps total à 0
                Ajouter le temps de recherche au temps total
                Afficher la moyenne de tous les essais
```

Le programme lui-même est simple; nous en présentons la plus grande partie avec quelques commentaires supplémentaires situés entre les divers sous-programmes. Notons que pour rester le plus possible dans la normale, puisque nous faisons des mesures, nous avons décidé d'utiliser le type `string` de la bibliothèque standard C++. Les sous-programmes de recherche sont ceux qui ont été vus plus tôt. Les résultats des tests étant la sortie critique du programme, nous les donnerons à la suite du programme et nous en tirerons quelques conclusions. Nous montrerons aussi comment rendre cette simulation plus réaliste.

```
const int TAILLEALPHA = 256;
const int PATRONMAX = 100;
const int PATRONMINI = 4;
const int PATRONMAXI = 15;
const int ESSAISMAX = 100;
const bool AFFICHE = false;
enum SorteRecherche
    {Recherche_Simple, Knuth_Morris_Pratt, Boyer_Moore, Rabin_Karp};
```

Nous avons utilisé un micro-ordinateur Ciara, dit compatible IBM, basé sur un processeur Pentium II à 385 MHz et tournant sous le système d'exploitation Windows 2000. Nous avons éliminé tous les processus sur lesquels nous avons le contrôle (quand on fait des mesures on doit toujours être conscient de la dépendance de ces mesures du système). Si ce dernier est un système permettant des processus simultanés, le temps mesuré est généralement un temps absolu qui comprend toutes les tâches effectuées au cours de ce laps de temps et donc pas nécessairement la tâche visée seule, ce qui fausse les mesures.

```
char LettreAleatoire()
// Engendre une lettre minuscule de "a" à "z" selon sa fréquence.
{
  int valeurAleatoire = entier_aleatoire(0, 1000);
  if(     valeurAleatoire <  93) return 'a';
  else if(valeurAleatoire < 107) return 'b';
  else if(valeurAleatoire < 141) return 'c';
  else if(valeurAleatoire < 174) return 'd';
  else if(valeurAleatoire < 333) return 'e';
  else if(valeurAleatoire < 347) return 'f';
  else if(valeurAleatoire < 362) return 'g';
  else if(valeurAleatoire < 371) return 'h';
  else if(valeurAleatoire < 455) return 'i';
  else if(valeurAleatoire < 460) return 'j';
  else if(valeurAleatoire < 463) return 'k';
  else if(valeurAleatoire < 516) return 'l';
  else if(valeurAleatoire < 545) return 'm';
  else if(valeurAleatoire < 617) return 'n';
  else if(valeurAleatoire < 649) return 'o';
  else if(valeurAleatoire < 673) return 'p';
  else if(valeurAleatoire < 681) return 'q';
  else if(valeurAleatoire < 746) return 'r';
  else if(valeurAleatoire < 825) return 's';
  else if(valeurAleatoire < 898) return 't';
  else if(valeurAleatoire < 960) return 'u';
```

```
  else if(valeurAleatoire < 980) return 'v';
  else if(valeurAleatoire < 981) return 'w';
  else if(valeurAleatoire < 984) return 'x';
  else if(valeurAleatoire < 985) return 'y';
  else return 'z';
} //LettreAleatoire;
```

Nous avons décidé de réaliser la fonction `LettreAleatoire` au moyen d'une grande instruction `if`. Nous avons pu éviter de vérifier les divers intervalles en vérifiant les valeurs en ordre croissant. Bien qu'il soit tentant d'utiliser une instruction `switch`, il vaut mieux éviter cette voie à cause du grand nombre de valeurs possibles pour la variable de contrôle `valeurAleatoire`. Il vaut toujours mieux éviter d'utiliser des instructions `switch`.

```
void MotAleatoire(string & Mot)
{
  // Engendre une suite de lettres aléatoires, basée sur des
  // fréquences données, de longueur suivante:
  // 1 lettre à 7 lettres : chacune avec une fréquence de 0.125
  // 8 lettres à 10 lettres : chacune avec une fréquence de 0.04166
  int valeurAleatoire, longueurMot;
  valeurAleatoire = entier_aleatoire(0, 7);
  if(valeurAleatoire == 0)
    longueurMot = 7 + entier_aleatoire(1, 3);
  else
    longueurMot = valeurAleatoire;
  Mot = "";
  for(int compteur = 0; compteur < longueurMot; compteur++)
    Mot += LettreAleatoire();
} //MotAleatoire;
```

On transforme le nombre aléatoire en un entier entre 0 et 7, de façon à ce que chaque nombre ait une fréquence de 12,5%. Nous utilisons une valeur nulle pour indiquer une longueur supérieure à 7, qui est alors choisie par un autre appel à `entier_aleatoire`, pour obtenir un nombre entre 8 et 10. Notez que les lettres aléatoires ainsi engendrées doivent être placées dans une chaîne de caractères vide à l'origine. Ce placement est fait par concaténation.

```
void EngendrerChaine(int LongueurChaine, string & Chaine)
// Engendrer une suite de "Mots" séparés par une seule espace de
// sorte que Chaine ait comme longueur LongueurChaine.
{
  string copie;
  Chaine = "";
  while(Chaine.size() < LongueurChaine){
    MotAleatoire(copie);
    Chaine += copie + ' ';
  } //while;
  if(Chaine.size() > LongueurChaine)
    Chaine.erase(LongueurChaine, Chaine.size()-LongueurChaine);
} //EngendrerChaine;
```

Les mots aléatoires suivis d'une espace sont ajoutés à la fin de la chaîne texte jusqu'à ce que la longueur voulue soit atteinte. Si le dernier mot ajouté rend la chaîne trop longue, on appelle `erase` pour réduire la chaîne à la longueur voulue en la tronquant.

```
bool VerifierAbsence(string &Patron, string Texte)
// Vérifier que Patron n'est pas une sous-chaîne de Texte; s'il l'est
// mettre Patron à la chaîne vide.
{ bool succes = Texte.find(Patron) == -1;
  if(! succes)
```

```
    Patron = "";
  return succes;
} //VerifierAbsence;

void ChoisirPatron(int LongueurPatron, string &Patron, string Texte)
// Engendrer un patron de longueur LongueurPatron qui ne soit PAS dans Texte.
{
  bool reponse;
  Patron = "";
  do{
    for(int compteur = 0; compteur < LongueurPatron; compteur++)
      Patron += LettreAleatoire();
    reponse = VerifierAbsence(Patron, Texte);
  }while(!reponse);
} //ChoisirPatron;
```

Nous engendrons un patron comme une suite de lettres aléatoires, puis nous vérifions que le patron ne se trouve pas dans la chaîne, car nous voulons tester nos algorithmes pour des recherches infructueuses. Nous avons utilisé la fonction `find` du type `string` pour effectuer cette recherche de patron. On doit absolument éviter les patrons d'un seul caractère, puisque des chaînes de bonne taille comprendront tous les caractères de l'alphabet et la procédure de choix de patron engendrerait des patrons à l'infini[7], en en cherchant un qui ne soit pas dans la chaîne.

Dans le programme principal, la chaîne texte est engendrée, la longueur du patron varie du minimum au maximum et la recherche est effectuée par les quatre algorithmes pour le nombre d'essais voulu. Nous pouvons accomplir cette tâche au moyen d'une boucle `for` qui comprend une instruction `switch`. Chaque fonction de recherche de patron est appelée de la même façon.

```
int main ()
{
  string patron, texte;
  int longueurTexte;
  float Donnees[4][PATRONMAXI-PATRONMINI+1][ESSAISMAX];
  int resultat;
  float tempsEcoule, tempsMoyen, tempsTotal;
  long top = clock();

  germe_aleatoire();
  // engendrer le texte à rechercher
  EngendrerChaine(20000, texte);
  longueurTexte = texte.length();
  // engendrer données de recherche
  for(int longueurMot = PATRONMINI; longueurMot <= PATRONMAXI; longueurMot++)
    for(int compteur = 0; compteur < ESSAISMAX; compteur++){
      ChoisirPatron(longueurMot, patron, texte);
      for(int recherche = Recherche_Simple; recherche <= Rabin_Karp;
          recherche++)
        switch(recherche){
        case Recherche_Simple :
          top = clock();
          resultat = RechercheSimple(patron, texte);
          tempsEcoule = clock() - top;
         Donnees[Recherche_Simple][longueurMot-PATRONMINI][compteur] =
                                                          tempsEcoule;
   ......
```

[7] Un cauchemar prolétaire.

On calcule enfin les résultats et on les affiche dans une table compacte. On pourrait placer le programme dans une boucle permettant à l'utilisateur de spécifier un nouveau germe pour la génération des nombres aléatoires pour chaque répétition de la boucle. Cependant, puisqu'on engendre 100 patrons pour chaque longueur, les résultats ci-dessous seront assez représentatifs.

Avant d'examiner la table des résultats, regardons une partie du texte engendré:

```
mvnmo are szsvne srlnud eraieuc tettu u anueesl vserssiu qapusl emenan olsflria
ue msrfaetne anatz uxaxi jeag lle elasi rmtis liuat altspidun nsllus ltfeoe e s
smts n erjt rszsd it suen ra anaeii tasqdt emldqaie ar non m adaz imxeendas ir e
besn intfn ciaa lua one uueetshmie en epvrzew ebv ap t nuiase iesu cana saeria n
 prlcciaee eaan r uecuereer d hso ineeo td e lse rpeleuu omuc maalemi ae ngncsaq
i cdfsu ipadsc kt aeiuoe oelntas auasten liesa utft rjio e c ecalani e niaen orv
ie ustieisru rqso aaletdlsud vdiv jeeqr riovcau sarr nteatr nue lhsueh velrdke t
rztn eo nvg dlrrvd ne tdn q ira ecida ocauaeujda ieto a f nuoraos aqmeopp n rsil
 tvke abtu sbre ntoend a nnaenaas imntras ete tledeee nti e apco dii lipaeflsas
tnuta lru lepalnj w htnhuvne e deeu s nlelzb vuuaanvne edu rnsea pitl tpomrd eze
euee lnuimii srnu p o iefhe e rdcv isaar s t s ri fsenue e ivr o tee srcc ns am
leresoz ee ntn a gcsmarap svzrm loit erero a n sc indtepn escairn aaajr alsnnf i
sezlil uadis elrn e olezgtne berlbs dirl o hne nmesmai anltp rl r steeu sqksgq t
mauum sdv srpavt ubee nel c daia rrsdeli n l rfasm armeoir sbelsur oedrld sau ei
ernqq eiraatc dieelntfhl e izia grselr kpa lansrim eapoaim ueaco eosi m ce fieur
 s suifl veeu atu iaeq zrqe nzisc olzsvg ti gnepclp essico nslzam n wtej a i ra
dmfinamlem atiise onn iorfu enmleoeote i lveqgu efe smt ettlu zaa rct ibi deet n
 riieaestla nja crtoanst adnnu iu cs t ronci eqq lanitnb dias t dmitarqg vanssc
aenetre reonsesaen celd aoasealrem qiespitas bse nseem ueli tsei aen tr azpf rre
 eotteiauce sao s asmubt eetta vs terianidp lrue etfs oso runpg enfmo ern vebun
hnbvlaget rlrsef eaobeelmpe uhee etueli teroaliqa uwesl sastug vuregzaasx aeahq
ec euasee raadfa uarcnf tua n uoncnae oee gtctea useaot ieiamueeli sapr ueardban
ra lqut aaranpsz m t ooc ibmn tssiec eauis e eioz cnnautn sctpee rtqv tuolp e di
sisr et ith os aseinuail ne onmui bnsi evrenou nfpuc epaeiee a eelpee qnfrrm r g
slt p stre ic saee envsimaa sutln retm rot iietnn vgfmieu isueu v tazeril la irn
eei raeetni reenoiv tipcualelz esl tassms ii cnnuu siussa i u es avcrea onnoai t
reinreaz i i aier cu lv tsecarn emsebn ugid tene ec zr tirsnti rbidvl mserpzu rd
utaae f celue euvharmu ovan rllihebe emden ufsuuezs azc np oitnt tnltvcsta trlen
lds aedseo aenuoa nneefuote mlfrei ie eiiuestp tt e edosdlr a v luelds rau tlc z
```

Voici un texte qui ne ressemble en rien à du français! On n'y reconnaît que quelques mots très courts, comme "rot" et "tua"; nous sommes passés très près de "armoire" avec "armeoir". Il est clair qu'il faudrait un temps extrêmement long à un anthropoïde installé à une machine à écrire pour produire une phrase française ayant un sens, sans parler des œuvres complètes de Victor Hugo! La génération de vrais mots français par utilisation de méthodes aléatoires est un problème intéressant et ardu. Les suites de lettres en français ne sont pas aléatoires; certaines suites, comme "*es*", apparaissent souvent, tandis que d'autres n'apparaissent jamais. L'étude de la cryptographie a permis de développer des tables de paires de lettres ou bigrammes (fréquences des paires de lettres) et de triplets de lettres ou trigrammes (fréquences des suites de trois lettres) pour permettre de casser des méthodes de chiffrement. Par exemple, parmi les bigrammes, les plus fréquents sont: *es, en, ou, de, nt, te, on, se, ai, it, le, et, me, er, em, oi, un* et *qu*. Mais il faut savoir que la plupart des probabilités associées à des bigrammes ont une valeur voisine de zéro. Malgré tout, l'utilisation de telles données permettra d'engendrer des mots qui ressembleront plus au français, mais même cette technique est loin de pouvoir nous permettre la production de phrases françaises sensées.

Une autre stratégie serait de construire des phrases à partir de mots choisis aléatoirement dans des listes de mots français connus. On pourrait aussi conserver un grand nombre de passages textuels pris dans la prose française et choisir les passages aléatoirement. Cependant, de tels efforts vont bien au-delà des objectifs de cette étude de cas dont les résultats sont présentés sous forme de table (voir la figure 3.16).

Ces temps incluent le pré-traitement, comme la construction des tables pour les algorithmes de Knuth-Morris-Pratt et Boyer-Moore. Trois des techniques, l'algorithme de Rabin-Karp, l'algorithme de la recherche simple et l'algorithme de Knuth-Morris-Pratt, ont une performance semblable pour toutes les longueurs de patron, tandis que l'algorithme de Boyer-Moore a une performance qui s'améliore avec la taille du patron. Ce résultat ne

devrait pas être surprenant, puisque la recherche va de droite à gauche dans le patron et laisse un plus grand nombre de caractères non examinés au fur et à mesure que le patron grandit. Le comportement devient en fait sous-linéaire.

```
Recherche sans succès dans un texte de longueur 20000
Résultats moyens basés sur 100 essais.
Patron Recherche Simple Knuth-Morris-Pratt  Boyer-Moore    Rabin-Karp
   4            4.30           4.32            2.20           6.61
   5            4.10           5.11            1.20           6.73
   6            3.60           4.60            2.71           5.42
   7            4.31           4.40            1.30           6.41
   8            4.40           4.71            1.20           6.61
   9            5.22           3.70            1.70           6.01
  10            4.32           3.90            1.20           7.21
  11            3.91           4.61            0.60           7.00
  12            3.20           5.41            0.80           6.21
  13            4.21           4.00            1.50           5.91
  14            4.20           4.10            1.10           6.32
  15            3.90           4.50            0.60           6.72
```

Figure3.16 Sortie du programme de mesure des algorithmes de recherche de patron

L'algorithme de Rabin-Karp est l'algorithme le plus lent, un résultat qui n'est pas surprenant non plus, étant donné la quantité de calcul numérique. La performance est à peu près constante, puisque les calculs sur le texte sont les mêmes quel que soit le patron. Par contre, il faut noter que l'algorithme de Rabin-Karp aurait certainement une meilleure performance pour un alphabet plus restreint. En particulier, pour un choix binaire de 0 ou de 1, les calculs numériques seraient bien plus simples et la performance serait améliorée de façon spectaculaire.

L'algorithme de la recherche simple a une performance semblable et peut-être même meilleure que l'algorithme de Knuth-Morris-Pratt, ce qui peut sembler a priori quelque peu surprenant. Cependant, on devrait tenir compte de la génération des données en analysant ces résultats. La recherche simple a une performance médiocre si la première différence se produit après les quelques premiers caractères. Comme nous l'avons déjà mentionné, certaines suites de caractères se produisent relativement plus fréquemment dans les textes français réels, tandis que la plupart des suites n'apparaissent jamais ou rarement. Donc, pour des textes français réels, l'algorithme de la recherche simple aura probablement des correspondances initiales plus fréquentes que pour des textes totalement aléatoires, comme nous en avons utilisé. On peut donc s'attendre à ce que la performance de l'algorithme de la recherche simple se dégrade avec des textes français réels, tandis que ce changement de textes n'aura aucun effet sur la performance de l'algorithme de Knuth-Morris-Pratt. Nous pouvons aussi prévoir une performance bien pire de la recherche simple pour un ensemble de caractères plus restreint, et, extrêmement mauvaise, si nous n'avons que deux caractères, comme 0 et 1.

Pour des patrons qui ont plus que quelques caractères, l'algorithme de Boyer-Moore est nettement gagnant, car la performance devient meilleure avec des longueurs de patron qui augmentent. On doit noter que l'algorithme de Boyer-Moore, comme l'algorithme de la recherche simple, bénéficie de l'aléatoire et de la variété des données qui produisent la différence rapidement. Cette performance se dégradera probablement avec un ensemble de caractères plus restreint ou avec des textes français réels.