Projet Algo

RECHERCHE DE MOTIFS

(Pattern Matching)

Sidi Mohamed Cheine

Stéphane Colson

Yann Magnabosco

Décembre 2000

TABLE DES MATIERES

I-INTRODUCTION 2

II-LES ALGORITHMES 3

1. L'algorithme naif 3
   1.1. Principe 3
   1.2. Algorithme 3
   1.3. Complexité 4
   1.4. Implantation en C 4
2. L'algorithme de Knuth-Morris-Pratt 4
   2.1. Principe 4
   2.2. Algorithme 5
   2.3. Complexité 6
   2.4. Implantation en C 6
3. Algorithme de Rabin-Karp 6
   3.1. Principe 6
   3.2. Algorithme 8
   3.3. Complexité 8
   3.4. Implantation en C 9
4. Algorithme de Boyer-Moore 9
   4.1. Principe 9
   4.2. Algorithme 10
   4.3. Complexité 11
   4.4. Implantation en C 12
5. Algorithme de Aho-Corasick 12
   5.1. Principe 12
   5.2. Algorithme 13
   5.3. Compléxité 14
   5.4. Implantation en C 14

III-ANALYSE DES RESULTATS 14

1. Test 1: motif de taille fixe - texte de taille variable (binaire) 15
2. Test 2: motif de taille fixe - texte de taille variable (texte) 16
3. Test 3: texte de taille fixe - motif de taille variable (binaire) 17
4. Test 4: texte de taille fixe - motif de taille variable (texte) 18
5. Test 5: recherche multiple 19
6. Test 6: mise en évidence du plus mauvais cas de l´algo naif 20

IV-CONCLUSION 21
Bibliographie 22
ANNEXES 23
Listings de naif, Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp, Aho-Corasick

I-INTRODUCTION

Trouver toutes les occurrences d'un motif dans un texte est un problème qu'on rencontre fréquemment dans les éditeurs de texte. Le plus souvent, le texte est un document en cours d'édition, et le motif recherché est un mot particulier fourni par l'utilisateur. Des algorithmes efficaces pour ce problème peuvent grandement améliorer la réaction du programme d'édition. Les algorithmes de recherche de motif sont également utilisés, par exemple pour trouver certains motifs dans des séquences d'ADN.

La compilation d'un langage possède aussi une phase centrale de recherche de motifs. Par exemple, pour vérifier la syntaxe d'un code source, le compilateur peut regarder si le code correspond à la grammaire formelle du langage. Cette étape est étroitement liée à la recherche de motifs.
Nous verrons que la taille de l'alphabet dont proviennent les caractères de la chaîne est un facteur important dans la conception de ces algorithmes de recherches de motifs.

Une opération fondamentale sur les chaînes est l'appariement de motifs (pattern matching), on formalise le problème de la recherche de motif de la manière suivant: on suppose que le texte est un tableau T [0.. n-1] de longueur n et que le motif est un tableau P [0..m-1] de longueur $m \leq n$ et que l´alphabet est noté Σ.

Les algorithmes que nous allons étudier ont une histoire intéressante que nous allons résumer pour permettre de replacer chaque méthode dans son contexte.

Il existe un algorithme naïf très répandu pour le traitement de chaînes.Bien qu'il s'accompagne d'une complexité dans le pire des cas proportionnel à m.n, les chaînes que l'on rencontre dans nombre de situations conduisent à des temps d'exécution presque toujours proportionnel à m+n. En outre, comme cette méthode peut bénéficier de la conception architecturale intelligente de beaucoup de machines, une version optimisée de l'algorithme constitue un standard difficile à battre.
En 1970, S.A.Cook démontra un résultat théorique concernant un type particulier de machine abstraite qui impliquait l'existence d'un algorithme permettant de résoudre le problème de l'appariement de motifs en temps proportionnel à m+n dans le pire des cas.
D.E.Knuth et V.R.Pratt suivirent avec patience le raisonnement que Cook avait élaboré pour sa démonstration (qu'il n'avait pas du tout conçu pour être une solution pratique) et parvinrent à un algorithme qu'ils purent affiner en une méthode pratique relativement simple.
C'est là un exemple rare et stimulant de résultat théorique qui s'accompagne d'une solution pratique immédiate. Mais il s'avéra que J.H.Morris avait découvert pratiquement le même algorithme.

Knuth , Morris et Pratt ne purent publier leur article avant 1976. Entretemps, R.S Boyer et J.S Moore avaient découvert un algorithme bien plus rapide dans beaucoup de situations, car il n'examine qu'une portion réduite des caractères de la chaîne de texte. Un grand nombre d'éditeurs de textes utilisent des variantes de cet algorithme.
En 1980, R.M. Karp et M.O Rabin s'aperçurent que le problème n'était pas aussi éloigné de celui d'une recherche standard qu'on avait voulu le croire et proposèrent un algorithme presque aussi simple que l'algorithme naïf, de temps d'exécution pratiquement toujours proportionnel à m+n. De plus leur algorithme s'adapte aisément à des motifs bidimensionnels, ce qui le rend

plus utile que les autres dans le traitement d'images.

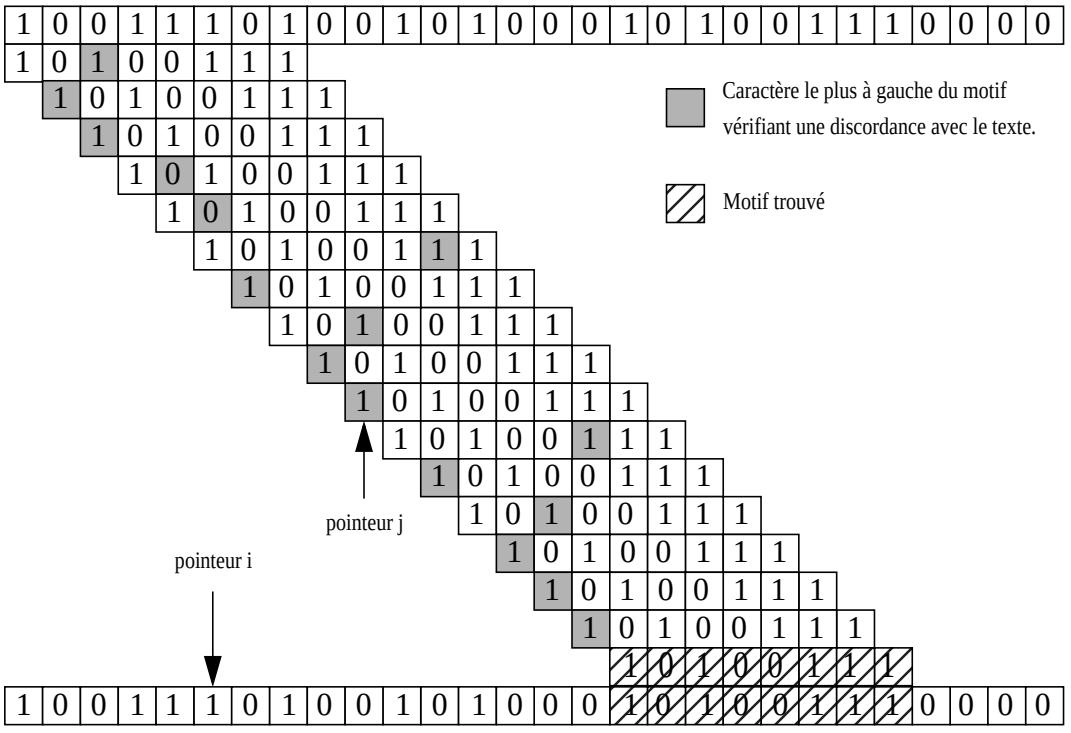
II-Les algorithmes

1. L'algorithme naif

1.1. Principe

La méthode qui vient immédiatement à l'esprit pour l'appariement des motifs consiste simplement à vérifier, pour chaque position du texte où le motif pourrait s'apparier, s'il y a véritablement concordance ou non. L'algorithme suivant effectue de cette façon une recherche de la première occurrence d'un motif p dans un texte t.

Exemple: on recherche le motif:10100111 dans le texte suivant



Le programme conserve un pointeur i dans le texte et un autre j dans le motif. Tant qu'ils font référence à des caractères concordants, les deux pointeurs sont incrémentés. Si i et j pointent sur des caractères discordants, j est remis à pointer sur le début du motif et i est repositionné de manière à faire avancer le motif d'un cran à droite en vue d'une nouvelle comparaison avec le texte.

1.2. Algorithme

```
RechercheNaive ( P,m,T,n )
  i=j=0
  pour  i = 0 à n-1
```

tant que $T[i] \neq P[j]$ *et* $j < m$
$i = i-j+1$
$j = 0$
fin tant que
incrémenter i et j
si $j = m$ *enregistrer la position du motif trouvée* ($= i-m$)
sinon continuer avec $j=0$
fin pour

1.3. Complexité

L´algorithme naif est acceptable si le premier caractère de la chaîne est rare dans le texte, ce qui évite les ¨faux départs¨, et dans ce cas il a une complexité en O(n+m), sinon elle est en O(n-m+1).m.

1.4. Implantation en C

Les structures de données utilisées pour le texte(T) et le motif(P) dans nos programmes sont des tableaux alloués de manière dynamique.

2. L'algorithme de Knuth-Morris-Pratt

2.1. Principe

L'idée essentielle dans cet algorithme est la suivante : lorsqu'une discordance est constatée, le ¨faux départ ¨ est composé de caractères connus à l'avance (puisqu'ils sont dans le motif). Nous devrions être capables de tirer profit de cette information au lieu de reculer le pointeur *i* par delà tout ces caractères connus.

Pour illustrer ce phénomène de manière simple, supposons que le premier caractère du motif n´apparaîsse qu´une seule fois dans celui-ci (disons 1000000).
Supposons maintenant que l´on ait un faux départ de longueur *j* à un certain endroit du texte. Lorsque la discordance est constatée, nous savons, en vertu du fait que *j* caractères concordent, qu´il est inutile de ¨reculer¨ le pointeur de texte *i*, puisque aucun des *j-1* caractères précédents du texte ne peut s´apparier au premier caractère du motif. L´idée de l´algorithme de Knuth-Morris-Pratt est de faire de telle sorte que le pointeur *i* ne soit jamais décrémenté.
Pour cela il calcule un tableau *suivant* qui permettra de déterminer de combien de positions nous devons reculer lorsqu´une discordance est détectée.

Pour $j>0$, la valeur de suivant[j] est le plus grand nombre $k<j$ tel que les *k* premiers caractères du motif se superposent aux *k* derniers de *j* premiers caractères du motif.
Ce tableau suivant permet d´éliminer le recul du pointeur de texte *i*.
Lorsque i et j pointent sur deux caractères discordants (alors que l´on cherche un appariement débutant en $i-j+1$ dans le texte), la prochaine position envisageable pour un appariement débute

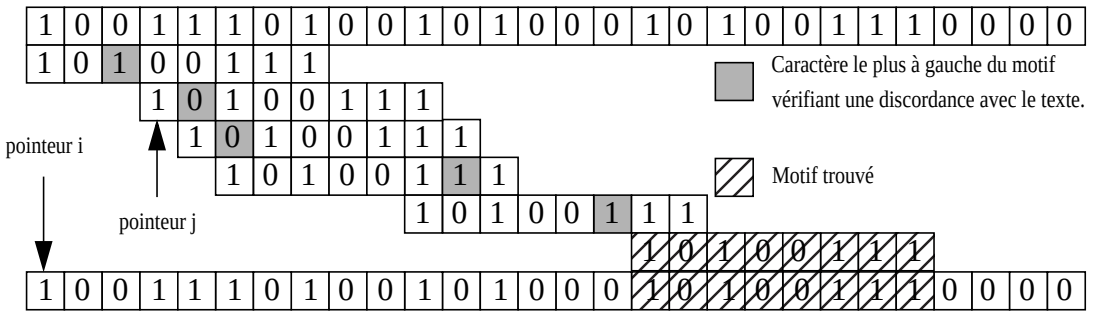
en i-suivant[j]. Mais, par définition de la table suivant, les suivant [j] premiers caractères après cette position sont nécessairement identiques aux suivant [j] premiers caractères du motif et il est donc inutile de faire reculer le pointeur si loin: il suffit de ne pas toucher au pointeur i et de positionner j en suivant j.

Formulation mathématique de ce tableau suivant:

suivant[j] = max{k / k<j, P[1..k-1] est un suffixe de P[1..j-1]}

Exemple:

Soit la recherche du motif: 10100111 dans un texte binaire composé de 0 et de 1.



Propriété:

L´algorithme de Knuth-Morris-Pratt pour la recherche de motif n´effectue jamais plus de m+n comparaisons de caractères.

Cette propriété est illustrée dans l´exemple ci-dessus et découle de manière évidente de l´écriture du programme: ou bien on incrémente *j*, ou bien on le repositionne à partir d´une valeur de la table suivant une fois pour chaque *i*.

Sur cet exemple, on voit que cette méthode effectue bien moins de comparaisons que la méthode naïve pour notre exemple binaire. Néanmoins, l´algorithme Knuth-Morris-Pratt ne sera pas beaucoup plus performant que la méthode naïve dans nombre de situations réelles, car peu d´applications amènent à rechercher des motifs hautement répetitifs dans des textes de même essence. Malgré tout, le fait que cette méthode ne ¨recule¨ jamais dans le texte en entrée constitue un avantage pratique fondamental et la rend très précieuse lorsque les recherches sont effectuées dans un gros fichier lu sur un support externe. (Dans ce cas, les algorithmes nécessitant un recul dans les données s´accompagnent de techniques complexes de gestion de tampons.)

2.2. Algorithme

```
RechercheKMP(p,m,t,n)
    InitialiserSuivant(p)
    Pour i = 0 à n-1 et j = 0 à m-1
      Tant que  j>=0  et  t[i] ≠ p[j]
```

```
    j = suivant(j)
      incrémenter i et j
  Si   j = m  le motif apparaît a la position i-m
  Sinon  continuer jusqu´à fin du texte
```

D´un point de vue fonctionnel, ce programme est identique à la recherche naïve, mais risque fort d´être bien plus efficace pour des motifs hautement répétitifs.

```
InitialiserSuivant (p,m)
    suivant[0] = -1
    j = -1
    Pour   i=0 à m-1
    Tant que   j≥0  et  p[i] ≠p[j]
        j=suivant(j)
    incrémenter i et j
    suivant(i)=j
```

Juste après que *i* et *j* aient été incrementés, il est acquis que les *j* premiers caractères du motif se répètent entre les positions *p[i-j-1]* et *p[i-1]* (les *j* derniers caractères parmi les *i* premiers du motif). Et j est le plus grand entier ayant cette propriété, car sinon une ¨concordance possible¨ du motif aurait été oubliée; ainsi, j est exactement la valeur à attribuer à *suivant[i]*.

2.3. Complexité

Cette procédure s'effectue en temps O(m) en moyenne ce qui conduit à une complexité globale en O(m+n), indépendante de la taille de l'alphabet. Cet algorithme a un bon comportement: il a la même complexité moyenne que tous les autres mais n'atteint pas O(mn) dans le pire des cas. En effet, le temps passé dans le ''tant que'' interne est au plus $\log_\phi(m)$ (où ϕ est le nombre d'or).

La complexité dans le pire des cas vaut donc $O(n.\log_\phi(m))$.
La complexité en espace est en O(n+2.m), ou n est la taille du tableau nécessaire pour stocker le texte et m est celle du tableau nécessaire pour stocker le motif.
On a deux fois m, car on utilise un tableau de taille m pour stocker les valeurs du tableau suivant.

2.4. Implantation en C

Pour le texte et le motif, on utilise les mêmes structures que précédemment. Ici, on utilise un tableau intermédiaire *suivant* alloué dynamiquement.

3. Algorithme de Rabin-Karp

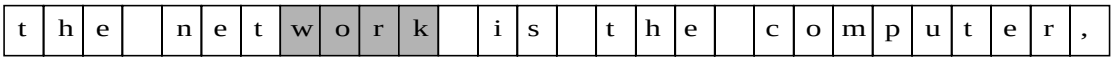
3.1. Principe

Il est possible d´améliorer l´algorithme naîf en utilisant un grand volume de mémoire. Celui-ci va traiter chaque suite de *m* caractères du texte comme clé d´une table d´adressage dispersé standard, et effectuer des comparaisons d´entiers.

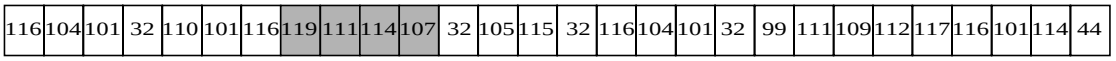
Le premier problème serait donc le stockage en mémoire de cette table, mais en fait on se rend compte qu´il n´est pas nécessaire de la conserver puisque le problème est organisé de maniére à effectuer une seule comparaison dans la table de hachage à la fois.
Ensuite, Rabin et Karp ont proposé une solution simple pour la fonction d´adressage que nous appellerons *h* telle que *h(x) = x modulo q* avec *q*, un grand entier premier.
Il est choisi très grand, ainsi la fonction *h* renverra un entier non nul différent pour chaque suite de *m* caractères, en veillant bien à ne pas dépasser la capacité de mémoire ordinateur lors de chaque calcul.

Exemple:

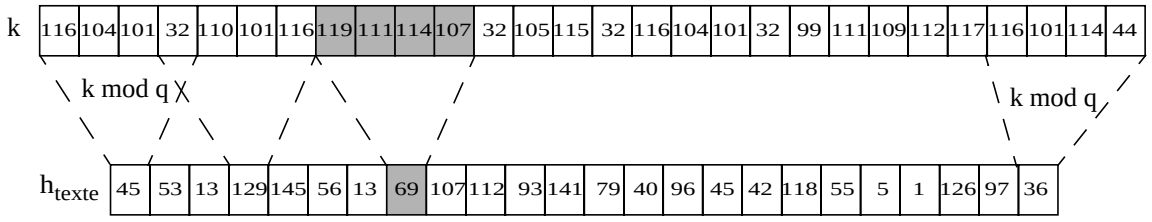
On recherche la chaîne de caractère *work* de taille 4 dans le texte suivant.



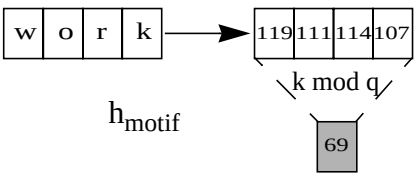
Le premier traitement, très simple en terme de calcul machine, consiste en la transformation des caractères en leur valeur décimale sur la table ASCII. Cela donne:



Nous transformons ensuite la chaîne de caractère et tous les blocs de taille 4 (taille du motif à rechercher) du texte à l´aide de la fonction *h*.
Pour cet exemple, nous avons préalablement fixé la taille de l´alphabet à 32 caractères (*d=32*) et le paramètre *q* à 149.



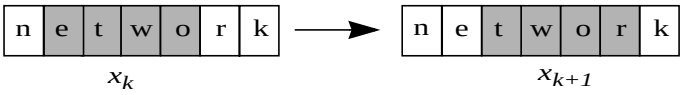
De même pour le motif work.



La fonction de hachage ramène la comparaison de m lettres à celle de deux entiers (on va rechercher 69 dans la table h).

Encore faut-il qu'elle ne soit pas trop difficile à calculer. Karp et Rabin ont utilisé la fonction $h(x) = x \bmod q$ pour un nombre premier q suffisamment grand. Si à $t_k \ldots t_{k+m-1}$ on associe $x_k = t_k.b^m + .. + t_{k+m-1}$ en considérant les lettres comme des entiers, alors on peut obtenir facilement x_{k+1} à partir de x_k:

$$x_{k+1} = (x_k - t_k.b^{m-1}).b + t_{k+m}$$



3.2. Algorithme

Recherche Rabin-Karp (motif,texte,positions)
initialiser h_{motif} *et* h_{texte} *à 0*
m = longueur du motif
n = longueur du texte
$dM = d^m$
pour i (0 --> m) faire
  $h_{motif} = (h_{motif}.d + Indice(Motif[i])) \text{ modulo } q$
  $h_{texte} = (h_{texte}.d + Indice(Texte[i])) \text{ modulo } q$
Fin pour
  Tant que j $< (n - m)$ *faire*
    $h_{texte} = (h_{texte} + d.q - Indice(Texte[i]).dM) \text{ modulo } q$
    $h_{texte} = (h_{texte}.d + Indice (Texte[i+m])) \text{ modulo } q$
    Si $(h_{texte} = h_{motif})$ *alors verifier si les caractères correspondent bien 1 a 1,*
      mémoriser position du motif trouve
    Fin Si
  Fin Tant que

La fonction Indice renvoie le code ASCII associe au caractère lu dans le motif ou dans le texte.

3.3. Complexité

La complexité associée à cet algorithme est en O(mn) dans le pire des cas et O(m+n) si tout va bien.
Le pire des cas ne sera jamais atteint, étant donné qu´on utilise une fonction de hachage renvoyant un entier fonction de la zone de texte testée. Cet entier sera unique sur l´ensemble du texte et nous n´aurons à tester les caractères qu´en cas d´égalite entre le motif et la zone de texte, d´où la complexité moyenne en O(m+n).

Le plus gros avantage de cet algorithme est qu´il s'adapte facilement à la recherche de plusieurs motifs . Ce qui peut aider à la recherche de texture dans les images, ou de séquences ADN. Nous le vérifierons dans le chapitre traitant des tests un peu plus loin.

3.4. Implantation en C

Nous avons utilisé des tableaux pour stocker le motif et le texte, avec allocation dynamique en fonction de la taille renvoyée par un appel à une structure *stat*.

4. Algorithme de Boyer-Moore

4.1. Principe

L´algorithme de Boyer-Moore ressemble étrangement à l´algorithme naïf, sauf qu´il compare le motif (P) au texte (T) de droite à gauche, et qu´il augmente le décalage s par une valeur qui n´est pas nécessairement égale à 1.

L´algorithme de Boyer-Moore intègre deux heuristiques, l´heuristique du mauvais caractère et l´heuristique du bon suffixe, qui lui permettent d´éviter une grande partie du travail effectué par l´algorithme naïf par exemple. Ces heuristiques permettent souvent à l´algorithme de sauter entièrement l´examen de nombreux caractères du texte. On peut considérer qu´elles agissent en parallèle et indépendamment. Lorsque deux caractères discordent, chaque heuristique propose une augmentation du décalage *s* qui ne fasse pas manquer un décalage valide. Notre algorithme choisit la plus grande des deux. Nous allons maintenant étudier de façon plus précise ces deux heuristiques.

a) L´heuristique du mauvais caractère

Lorsque l´on trouve une discordance, l´heuristique du mauvais caractère utilise des informations sur la position du mauvais caractère $T[s+j]$ dans le motif pour proposer un nouveau décalage. On va augmenter *s* de la valeur $j-k$, où *k* est le plus grand indice de l´intervalle $0 \le k \le m-1$ tel que $T[s+j] = P[k]$, s´il existe.
Dans le cas où $k>j$, l´heuristique du mauvais caractère propose de décroître *s*. Cette recommandation sera ignorée par l´algorithme de Boyer-Moore, puisque l´heuristique du bon suffixe proposera un décalage positif dans tous les cas.
La fonction Calcul-Dernière-Occurrence, définie dans l´algorithme ci-après, implémente l´heuristique du mauvais caractère, elle définit $\lambda[a]$ comme étant l´indice de la position la plus à droite à laquelle apparaît le caractère *a* dans le motif, et ce pour chaque caractère de l´alphabet *(*On notera Σ l´alphabet et $|\Sigma|$ sa cardinalité*)*. Si le caractère n´est pas dans le motif, $\lambda[a]$ prend la valeur -1 (dans ce cas on augmente *s* de $j+1$).

b) L´heuristique du bon suffixe

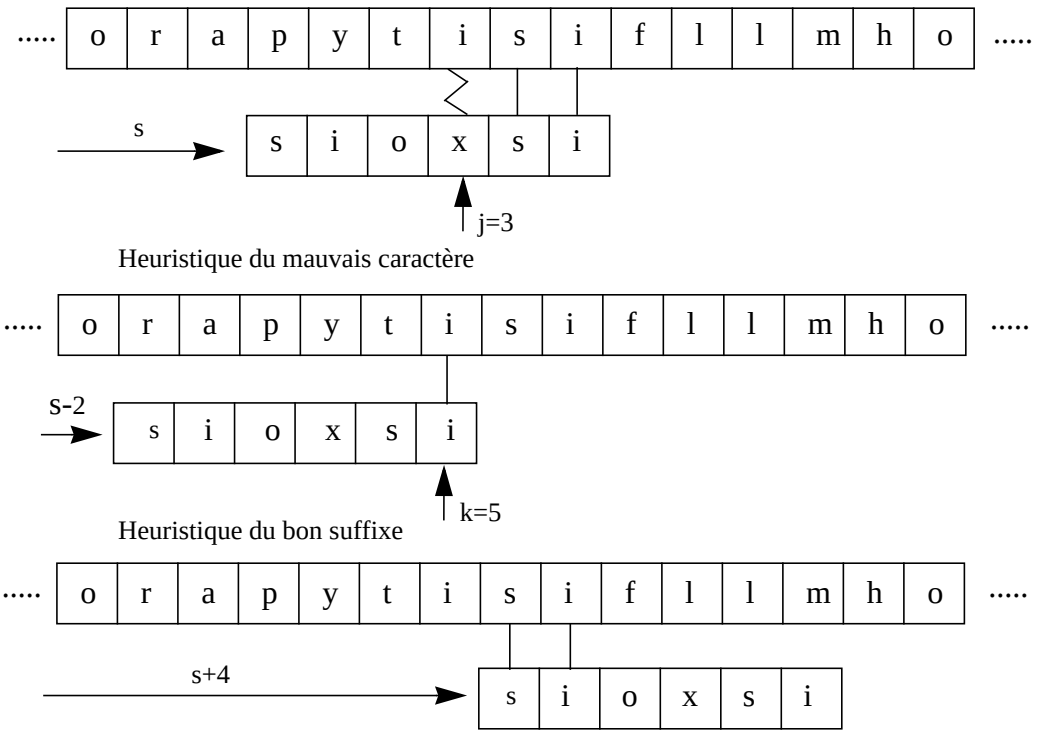
Si l´on trouve une discordance telle que $P[j] \neq T[s+j]$ où $j<m$, l´heuristique du bon suffixe propose d´avancer *s* de $\gamma[j]$ qui est la plus petite quantité par laquelle on préserve l´appariement des caractères du bon suffixe $T[s+j+1..s+m-1]$ avec des caractères du motif.
On peut définir $\gamma[j]$ de la manière suivante:

$$\gamma[j] = min(\{m-1 - \pi[m-1]\} \cup \{l - \pi'[l] : 0 \leq l < m \text{ et } j=m-1 - \pi'[l]\})$$

avec π et π' les fonctions InitialiserSuivant des motifs P et P´ où P´ est l´image miroir de P (*P´[m-1]=P[0], P´[m-2]=P[1],.....,P´[0]=P[m-1]*).

La fonction Calcul-Bon-Suffixe décrite au paragraphe 4.2, est une implémentation directe de l´équation précédente.

Exemple:



Dans cet exemple, on trouve une discordance à la position *j=3*.
L´occurrence la plus à droite du mauvais caractère (i) dans le motif se trouve à la position *k=5*, l´heuristique du mauvais caractère propose un décalage de *j-k=-2* caractères.
Avec l´heuristique du bon suffixe, un déplacement du motif de 4 positions vers la droite permet de garantir que tous les caractères du motif, correspondant au bon suffixe *(si)* trouvé dans le texte, correspondra à ces caractères du suffixe.
Comme l´heuristique du mauvais caractère propose un déplacement vers la gauche de 2 positions, ce qui est inférieur à la proposition de l´heuristique du bon suffixe, l´algorithme de Boyer-Moore augmente le décalage de 4 positons.

4.2. Algorithme

Recherche Boyer-Moore(T,P)
n = longueur(T)

```
m = longueur(P)
λ = Calcul-Dernière-Occurence(P, Σ)
γ = Calcul-Bon-suffixe(P)
s = 0
tant que  s ≤ n-m
   faire  j = m-1
      tant que  j ≥ 0  et  P[j] = T[s+j]
         faire  j = j-1
      si  j = -1
         alors  ¨Le motif apparaît avec le décalage¨  s
      sinon  s = s + max(γ[j] , j-λ[T[s+j]])


Calcul-Dernière-Occurrence(P,Σ)
   pour  chaque caractère a ∈ Σ
      faire  λ[a] = -1
   pour  j = 0  à  m-1
      faire  λ[P[j]] = j
   retourner λ


Calcul-Bon-Suffixe(P)
   π = InitialiserSuivant(P)
   pour  i = 0  à  m-1
      faire  P´[i] = P[m-1-i]
   π´ = InitialiserSuivant(P´)
   pour  j = 0  à  m-1
      faire  γ[j] = m - 1 - π[m-1]
   pour  l = 0  à  m-1
      faire  j = m - 1 - π´[l]
         si  γ[j] > l - π´[l]
            alors  γ[j] = l - π´[l]
   retourner γ
```

4.3. Complexité

Le temps d´exécution de la procédure Calcul-Dernière-Occurrence(P,Σ) est en O(m+|Σ|).
Celui de la fonction Calcul-Bon-Suffixe(P) est en O(m).
Dans le plus mauvais des cas, le temps d´exécution de recherche de l´algorithme de Boyer-Moore sera en O((n-m+1)m+|Σ|) puisqu´il faudra un temps en O(m) pour valider chaque décalage.

Le résultat en moyenne n/m peut être établi pour différents modèles de chaînes aléatoires, mais ces derniers sont souvent irréalistes. En pratique, seulement une poignée de caractères d´alphabet apparaissent dans le motif et chaque comparaison conduit à une translation de m positions.

La complexité en espace est en O(n+m+|Σ|).

4.4. Implantation en C

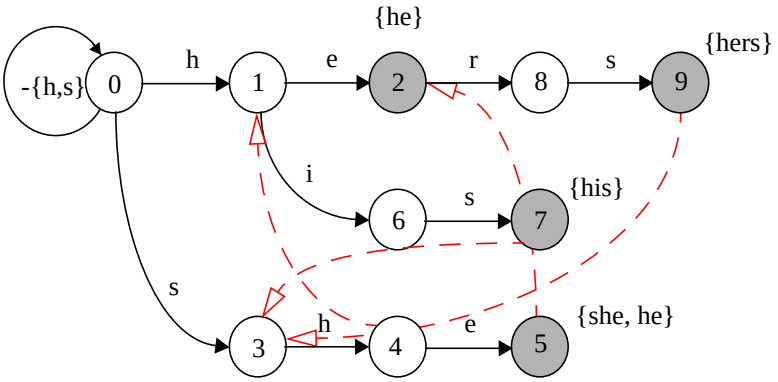
Pour le texte et le motif nous utilisons les mêmes structures de données que précédemment. En ce qui concerne les tableaux π, π´, γ, λ et P´, nous les avons alloués dynamiquement, leur taille est égale à la longueur du motif recherché.

5. Algorithme de Aho-Corasick

5.1. Principe

Plutôt que de faire la recherche des occurrences des motifs de l'ensemble $P=\{P_0,P_1,....,P_{k-1}\}$ dans le texte *T* séquentiellement, on peut la faire en ''parallèle''. Ceci constitue l'algorithme de Aho-Corasick qui peut être vu comme une généralisation de l´algorithme de Knuth-Morris-Pratt pour faire une recherche parallèle.

Exemple:



Dans l´exemple ci-dessus, nous avons construit l´automate pour rechercher les motifs suivants: { *he, she, his, hers* }.
L´etat 0 désigne l´etat de départ.
La fonction *g* (les flèches pleines) permet de passer d´un état à un autre pour une lettre donnée du texte dans lequel on fait une recherche (lorsque l´on est dans l´etat 2 et que l´on rencontre le caractère *r* dans le texte, on passe dans l´etat 8). L´absence de flèche partant d´un état vers un autre pour un caratère donné indique un échec. A ce moment là, on consulte la fonction *f* (les flèches en pointillés).
Supposons que l´on soit dans l´etat 4 et que l´on rencontre un caractère autre que *e*, dans ce cas la fonction *f* nous renvoie à l´etat 1 dans l´automate, en effet dans l´etat 1, on avait aussi rencontré le caractère *h*.
Lorsqu´on arrive sur un état représenté en gris, cela signifie que l´on a trouvé un ou plusieurs motifs (par exemple l´etat 5 indique qu´une occurence des motifs *she* et *he* a été trouvée).

5.2. Algorithme

```
Recherche Aho-Corasick(T,P)
    n = longueur(T)
    g = calculg(P,sortie)
    f = calculf(g,sortie)
    etat = 0
    pour  i = 0  a  n-1
        faire  tant que g(etat,T[i]) = echec
                    etat = f(etat)
               fin tant que
               etat = g(etat,T[i])
               si sortie(etat) ≠ vide
                    alors  "Le motif sortie(etat) apparaît a la position i"
               fin si
    fin pour


Calculg(P,sortie)
    nouveletat = 0
    k = nombre de motifs
    pour i = 0  a k-1
        faire  entrer(P_i)
    fin pour
    pour chaque lettre a tel que g(0,a) = echec
        faire  g(0,a) = 0
    fin pour


Entrer(P_i)
    m = longueur(P_i)
    etat = 0
    j = 0
    tant que g(etat,P_i[j]) ≠ echec
        faire   etat = g(etat,P_i[j])
                j = j+1
    fin tant que
    pour p = j  a  m-1
        faire   nouveletat = nouveletat+1
                g(etat,P_i[p]) = nouveletat
                etat = nouveletat
    fin pour
    sortie(etat) = P_i


Calculf(g,sortie)
    queue = vide
    pour chaque lettre a tel que g(0,a) = s ≠ 0
        faire  queue = queue U {s}
```

```
            f(s) = 0
fin pour
tant que  queue ≠ vide
   faire  prendre r dans queue
            queue = queue - {r}
            pour chaque lettre a tel que g(r,a) = s ≠ echec
               faire  queue = queue U {s}
                      etat = f(r)
                      tant que g(etat,a) = echec
                         faire  etat = f(etat)
                      fin tant que
                      sortie(s) = sortie(s) U sortie(f(s))
             fin pour
fin tant que
```

5.3. Compléxité

Le temps d´exécution des procédures *Calculg* et *Calculf* est proportionnel à la somme des longueurs des motifs (*M*), c´est à dire en O(M).
Le temps total d´exécution de l´algorithme de Aho-Corasick sera en O(n+M) puisque l´on considère les caractères du texte les uns après les autres.

5.4. Implantation en C

Nous avons implanté *g* et *sortie* de manière statique sous la forme d´un tableau à 2 dimensions avec un nombre de lignes correspondant au nombre d´etats de l´automate et un nombre de colonnes correspondant au nombre de lettres de l´alphabet pour *g* et au nombre de motifs de l´automate pour *sortie*.
La fonction *f* est implantée à l´aide de tableaux dont la taille dépend du nombre d´etats de l´automate.

III-Analyse des résultats

Pour les algorithmes *Naif*, *Boyer-Moore*, *Knuth-Morris-Pratt*, nous avons développé trois versions:
1- le motif est entré au clavier
2- le motif est lu dans un fichier
3- le motif est lu dans un fichier et on recherche toutes les occurrences de ce motif.
Pour les tests suivants, nous avons utilisé la version numéro 2.

Pour *Rabin-Karp* et *Aho-Corasick*, une seule implantation, correspondant à la version 3, a été développée en vue des tests.
L´algorithme d´*Aho-Corasick* n´a été testé que pour la recherche de plusieurs motifs, car pour une recherche unique, il a peu d´intérêt étant basé sur le principe de la construction d´un auto-

mate de recherche multiple.

Tous les tests ont été réalisés avec une station Sun Sparc Ultra 5, à l´aide de motifs ne se trouvant pas dans le texte pour faire un parcours total de celui-ci.
Nous avons également testé les fonctions Unix *strfind* et *fgrep* pour les comparer à «nos» algorithmes.
Dans toute la suite, les notations suivantes seront utilisées:

m = Longueur du motif
n = Longueur du texte
M = Millions de caractères

Les temps de recherche sont exprimés en secondes (s) et parfois en minutes (min).
On a utilisé l´appel bibliotèque fgetc pour remplir les tableaux correspondants au motif et au texte, on aurait pu utilisé fread.
Cela n´a pas d´incidence sur nos résultats, en effet lors des différents tests nous avons verifé les temps système qui étaient négligeables par rapport aux temps utilisateur.

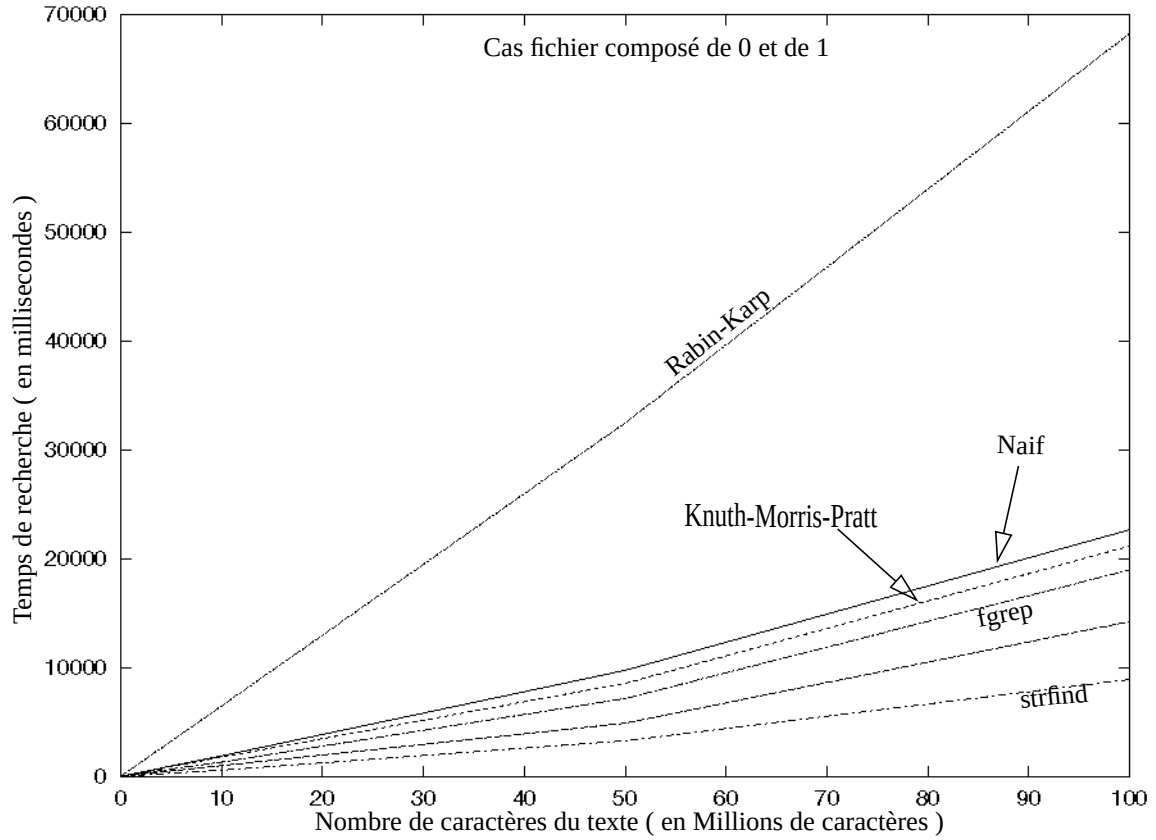
1. Test 1: Cas binaire - m=25 , n est variable

Le tableau 1 dresse les résultats des différents algorithmes, pour un motif composé de 1 et de 0 d´une longueur de 25, sur un texte binaire composé de 1 et de 0 dont la longueur *n* varie entre 100000 et 100 Millions de caractères.

Tableau 1: Binaire - Recherche d´un motif de taille 25 dans un texte de taille variable

Nombre de caractères texte	n=100000	n=500000	n= 1 M	n=10 M	n=50 M	n=100 M
naif	~ 0 s	0.1 s	0.2 s	1.9 s	9.8 s	22.7 s
B - M	~ 0 s	~ 0 s	0.1 s	1.0 s	4.9 s	14.3 s
K-M-P	~ 0 s	0.1 s	0.2 s	1.8 s	8.5 s	21.0 s
Rabin-Karp	~ 0 s	0.3 s	0.7 s	6.5 s	32.5 s	68.0 s
Fgrep	~ 0 s	~ 0 s	0.1 s	1.3 s	7.0 s (19 mn)	19 s
Strfind	~ 0 s	~ 0 s	~ 0 s	0.6 s	3.4 s	9.0 s

Figure 1



Interprétation du test 1

Dans ce cas, l´ensemble des algorithmes a un bon comportement. Boyer-Moore est néanmoins le plus rapide de tous car il évite bien des tests inutiles. La fonction strfind utilise Boyer-Moore dans son implémentation et obtient la palme de la rapidité de recherche. En ce qui concerne la fonction système fgrep, elle n´est plus très efficace sur des textes de taille supérieure à 10 Millions de caractères.

L´algorithme de Rabin-Karp apparaît bien plus lent que les autres car un lourd traitement sur le texte et le motif est à faire à chaque recherche lancée.

2. Test 2: Cas texte - m=25 , n est variable

Le tableau 2 dresse les résultats des différents algorithmes, pour un motif d´une taille de 25 caracteres, sur un texte d´un alphabet de 80 caractères dont la taille n varie comme suit.

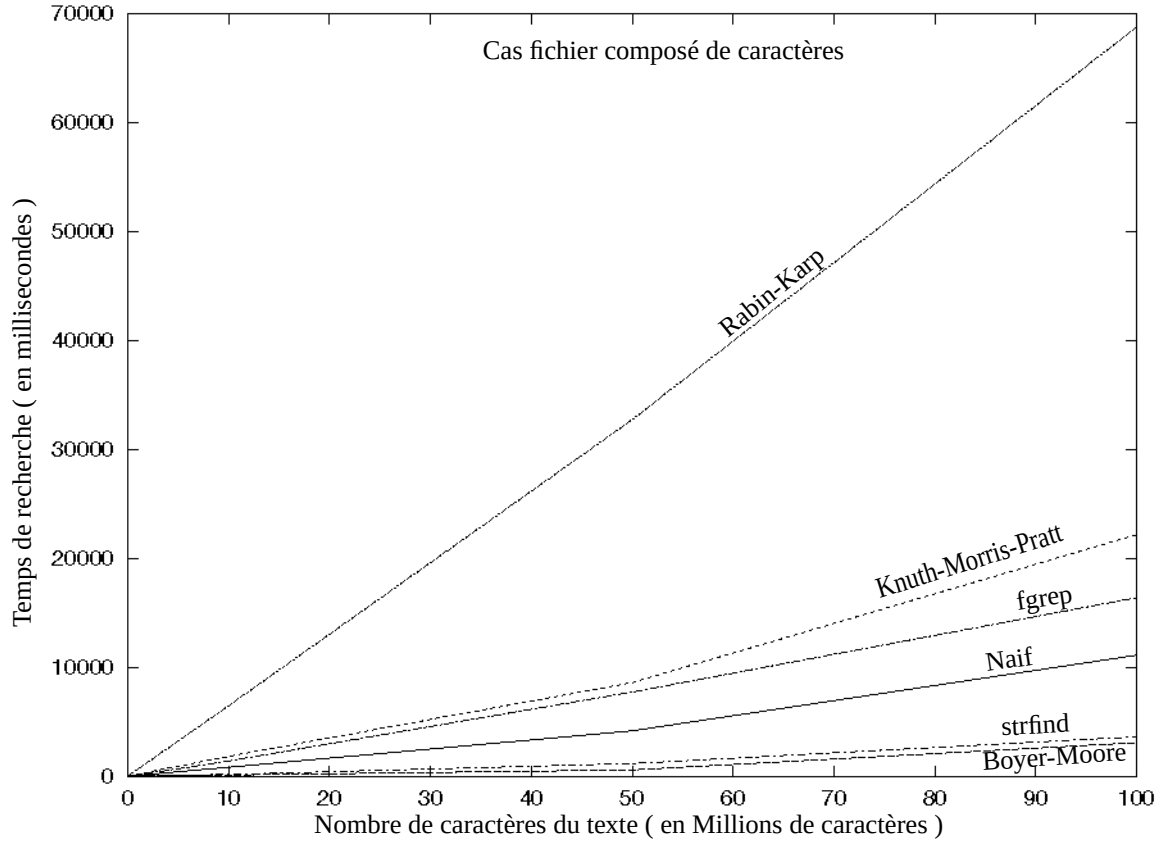
Tableau 2: Texte - Recherche d´un motif de taille 25 dans un texte de taille variable

Nombre de caractères texte	100000	500000	1 Million	10 Millions	50 Millions	100 Millions
naif	~ 0 s	~ 0 s	~ 0 s	0.8 s	4 s	11 s
B - M	~ 0 s	~ 0 s	~ 0 s	0.1 s	0.6 s	3 s
K-M-P	~ 0 s	~ 0 s	0.2 s	1.9 s	8.6 s	22 s

Tableau 2: Texte - Recherche d´un motif de taille 25 dans un texte de taille variable

Rabin-Karp	~ 0 s	~ 0 s	0.6 s	6.5 s	32.8 s	69.8 s
Fgrep	~ 0 s	0.1 s	0.1 s	1.4 s	7.7 s	16.4 s
Strfind	~ 0 s	~ 0 s	~ 0 s	0.2 s	1.2 s	3.6 s

Figure 2



Interprétation du test 2

Par rapport au premier test, la recherche dans un fichier texte est sensiblement plus rapide pour les algorithmes suivants: naif (2 fois plus rapide), Boyer-Moore (5 fois plus rapide) et strfind (3 fois plus rapide).
Ceci s´explique par le fait qu´un alphabet simple comme celui utilisé dans le premier exemple (0 ou 1) engendre beaucoup plus de «faux-départs». Les autres ont un comportement identique et un temps de recherche égal

3. Test 3: Cas binaire - m est variable , n = 10 Millions de caractères

Dans le tableau 3, on a fait changer la taille du motif composé de 0 ou de 1 en gardant cette fois-ci le même texte de taille 10 Millions de caractères.

Tableau 3: Binaire - Recherche d´un motif de taille variable dans un texte de taille 10 Millions de caractères

Nombre de caractères motif	10	500	1000	100000	500000	1 Million
naif	2.6 s	2.6 s	2.6 s	2.6 s	2.6 s	2.6 s
B - M	0.4 s	0.6 s	2.3 s	61 s	34 min	-
K-M-P	1.6 s	1.7 s	1.8 s	1.7 s	2 s	2 s
Rabin-Karp	6 s	6 s	6 s	6 s	6 s	6 s
Strfind	0.2 s	0.6 s	0.6 s	0.6 s	0.6 s	0.6 s
Fgrep	1.4 s	1.4 s	1.4 s	1.4 s	1.4 s	1.4 s

Interprétations du test 3

On remarque que les temps de recherche sont constants lorsqu´on fait varier la taille du motif, sauf pour l´algorithme de Boyer-Moore.
Ceci s´explique par le fait que les algorithmes ne testent jamais la totalité du motif, ils s´arrêtent dés qu´ils rencontrent une discordance.
Par contre Boyer-Moore nécessite des prétraitements proportionnels à la taille du motif, ce qui explique qu´il soit si long pour un motif de 500000 caractères.

4. Test 4: Cas texte - m est variable , n = 10 Millions de caractères

Dans le tableau 4, on utilise un alphabet de 80 caractères et on recherche des motifs générés aléatoirement de taille *m* variable entre 10 et 1 Millions de caractères.

Tableau 4: Texte: Recherche d´un motif de taille variable dans un texte de taille 10 Millions de caractères

Nombre de caractères motif	10	500	1000	100000	500000	1 Million
naif	1 s	1 s	1 s	1 s	1 s	1 s
B-M	0.4 s	~ 0 s	0.2 s	61 s	2053 s	-
K-M-P	2.0 s	2.0 s	2.0 s	2.0 s	2.0 s	2.0 s
Rabin-Karp	5.0 s	5.0 s	5.0 s	5.0 s	5.0 s	5.0 s

Interprétation du test 4

Les conclusions sont les mêmes que pour le test 3 sur des textes binaires.

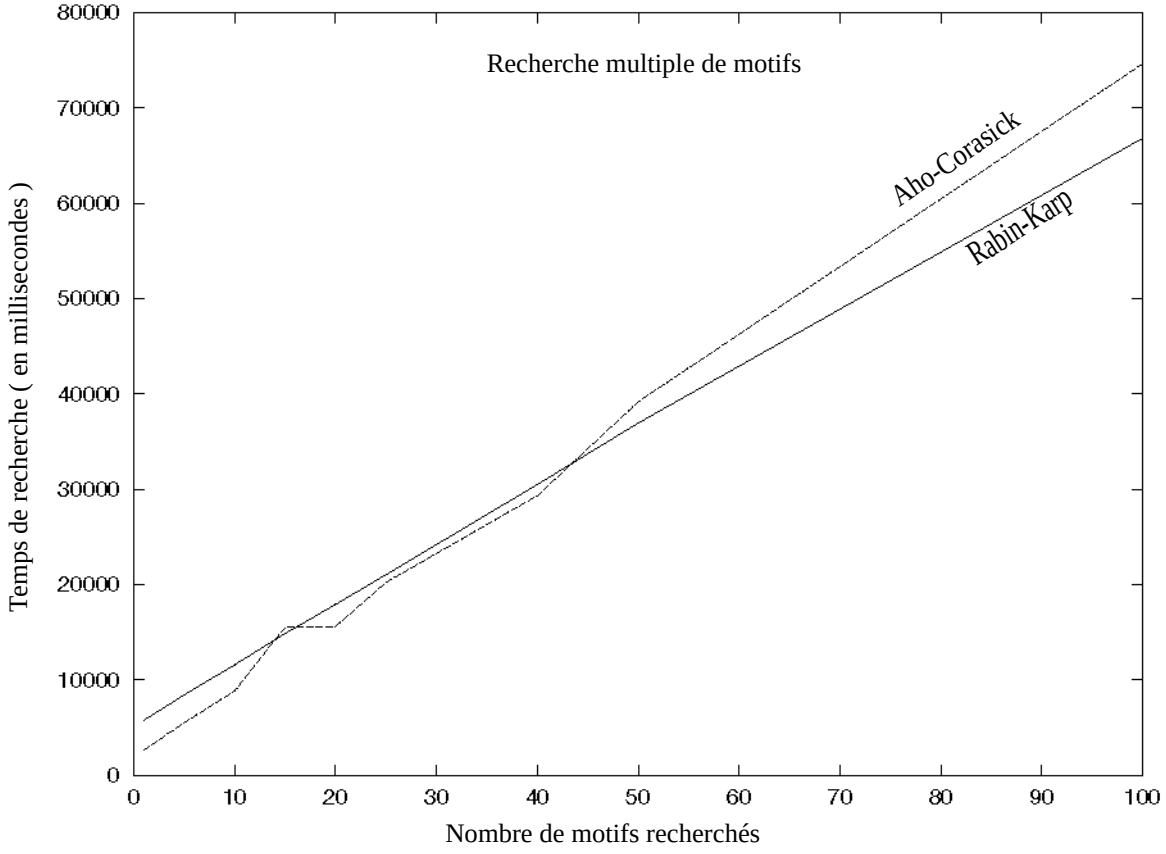
5. Test 5:Cas texte-Recherche multiple de motifs de taille 10 dans un fichier de taille 50 M

Le tableau 5 montre la recherche de plusieurs motifs (entre 1 et 100 motifs recherchés conjointement) de taille 10 caractères dans un fichier texte de taille 50 Millions de caractères.

Tableau 5: Recherche multiple de motifs texte de taille 10 dans fichier de taille 10M

Nbre motifs	1	5	10	15	20	25	40	50	100
Rabin-Karp	5.6 s	8.2 s	11.3 s	14.4 s	16.8 s	20.4 s	30.0 s	36.0 s	66.8 s
Aho-Corasick	2.7 s	5.7 s	9.2 s	12.7 s	17.0 s	20.6 s	29.9 s	40.1 s	74.6 s

Figure 3



Interprétation du test 5

La pente de la courbe décrivant le comportement de l´algorithme de Rabin-Karp en fonction du nombre de motifs recherchés est de 0.6 respectivement 0.7 pour Aho-Corasick . Ce qui signifie que lorsqu´on multiplie le nombre de motifs à rechercher par 100, on aura un temps de recherche de 60 fois supérieur pour Rabin-Karp (respectivement 70 pour Aho-Corasick). Pour tous les autres algorithmes (naif, Knuth-Morris-Pratt et Boyer-Moore), le temps de

recherche est directement multiplié par le nombre de motifs à rechercher.

6. Test 6

Le tableau 6 met en évidence la complexité dans le pire des cas de l´algorithme naif qui est en O(n.m). Pour cela, nous avons généré un fichier de 50 millions de caractères 1, et on recherche un motif de longueur 400 formé uniquement de 1 sauf le 400ème caractère qui est un 0.

Tableau 6:

	50 M
naif	34 min
Boyer-Moore	7.7 s
K-M-P	1.6 s
Rabin-Karp	31.6 s

La recherche naive va comparer dans ce cas tous les caractères du motif (m) avec tous les caractères du texte (n-m+1).
On peut remarquer au passage que l´algorithme Knuth-Morris-Pratt a la complexité la plus faible et ceci est dû au fait qu´il effectue dans le pire des cas (n+m) comparaisons (voir propriété ci-dessus).

IV- CONCLUSION

Les algorithmes de recherche de motifs ont en moyenne un temps d´exécution linéaire, ce que nous avons réussi à mettre en évidence dans le chapitre consacré à l´analyse des résultats.

En pratique, les algorithmes Rabin-Karp et Aho-Corasick sont utilisés pour faire des recherches de motifs en parallèle(ou recherche multiple) alors que les algoritmes naif, Knuth-Morris-Pratt et Boyer-Moore sont plus adaptés à rechercher un seul motif, sinon s´il y en a plusieurs la recherche est séquentielle.

Tableau 7: Récapitulatif des complexités

Algorithme	moyenne	maximale
Naif	**O(n)**	**O(m.n)**
K-M-P	**O(m+n)**	**O(n$\log_{\phi}$(m)**
B-M	**O(n/m)**	**O(n+m)**
R-K	**O(m+n)**	**O(m.n)**
A-C	**O(m+n)**	**-**

Bibliographie

1. T. Cormen, C. Leiserson, R. Rivest, Introduction à l'algorithmique DUNOD

2. Robert Sedgewick, Algorithmes en language C. Addison-Wesley Europe Intereditions, 1991.

3. Mikhail J. Atallah, Algorithms ant Theory of Computation Handbook.

4. Alfred V. Aho and Margaret J. Corasick, Efficient String Matching: an aid to bibliographic search. Communications of the ACM, 18, 333-340, 1975.

5. *http://sunburn.informatik.uni-tuebingen.de/~buehler/AC/AC.html* : Simulation d´une recherche avec l´algorithme de Aho-Corasick.

6. Roland Jegou - Cours d´algorithmie - Ecole Nationale Supérieure des Mines de Saint Etienne, (2000-2001)

7. Knuth, D.E., J.H. Morris and V.R. Pratt, Fast pattern matching in strings , SIAM J.Comput 6(2) (1977) 323-350.

8. Karp R.M. and Rabin M.O., Efficient randomized pattern-matching algorithms. IBM J. Res. Dev., 31, 249-260, 1987

9. Boyer R.S. and Moore J.S., A fast string Searching Algorithm - Communications of the ACM - Vol 20 number 10 (october 1977)

10. Rapport de DEA -Traitement de données textuelles - Samuel Devulder - *http://users.info.unicaen.fr/~devulder/dea/node1.html* (1993)