Algorithmique du texte

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Recherche dans un texte

Principe de la recherche

Problème

Soit Σ un ensemble fini de caractères, appelé **alphabet**.

On s'intéresse ici au problème suivant :

- **Entrées :**
  - une chaîne de caractères s sur l'alphabet Σ ;
  - une autre chaîne de caractères m sur Σ, appelée **motif**, telle que $|m| \leq |s|$.
- **Sortie :** un résultat partiel correspondant à l'indice de la première occurrence de m dans s s'il est présent.

Principe de la recherche

Remarque

L'hypothèse que Σ soit fini va mener à certaines optimisations, car le nombre de caractères de Σ sera négligeable devant la taille des chaînes de caractères.

On parle alors d'**algorithmique du texte** pour désigner des algorithmes tirant partie de cette contrainte sur les données.

Principe de la recherche

Exemple

Voici trois alphabets très importants, avec lesquels les algorithmes de ce chapitre sont très utilisés :

- $\Sigma = \{\text{caractères } \textbf{ASCII}\}$;
- $\Sigma = \{0, 1\}$, ce qui permet de travailler sur des recherches en **binaire** ;
- $\Sigma = \{A, T, C, G\}$, correspondant aux bases d'un brin d'**ADN**, et qui ouvre la porte à beaucoup d'applications en **bio-informatique**.

Algorithme naïf en force brute

Brute force

Une solution naïve consiste à parcourir chaque position de s afin de tester si le motif est présent à partir de cette position.

Algorithme naïf en force brute

OCaml

```
exception Trouve of int ;;
exception PasDeMotif ;;

let cherche_motif m s i =
  let p = String.length m in
  try
    for j = 0 to p-1 do
      if s.[i+j] <> m.[j]
      then raise PasDeMotif
    done;
    true
  with PasDeMotif -> false
;;

let recherche_naive m s =
  let n = String.length s in
  let p = String.length m in
  try
    for i = 0 to n-p do
      if cherche_motif m s i
      then raise (Trouve i)
    done;
    None
  with Trouve i -> Some i
;;
```

Algorithme naïf en force brute

C

```c
#include <string.h>

/* recherche_naive(m,s) recherche le motif m dans la chaine
 * s et renvoie l'indice de la première occurrence s'il est présent
 * ou -1 sinon */
int recherche_naive(const char *m, const char *s)
{
  int n = strlen(s);
  int p = strlen(m);
  for (int i = 0; i <= n-p; i++)
  {
    int j;
    for (j = 0; j < p; j++)
    {
      if (s[i+j] != m[j])
        break;
    }
    if (j == p)
      return i;
  }
  return -1;
}
```

Algorithme naïf en force brute

Complexité

La complexité temporelle de cet algorithme dans le pire des cas correspond au maximum de comparaisons, c'est à dire $\mathcal{O}(np)$. Mais on peut remarquer qu'il est assez difficile d'obtenir un exemple concret, ce qui fait penser que ce pire cas est **rare**.

Exemple

Le pire cas est atteint avec $s = \underbrace{aa\ldots a}_{n \text{ fois}} = a^n$ et $m = a^{p-1}b$.

Algorithme naïf en force brute

Complexité en pratique

Ce qui va se passer dans une application usuelle de cet algorithme, c'est qu'au bout d'une ou deux comparaisons, on pourra invalider la position et passer à la suivante.

On va alors avoir une complexité en $\mathcal{O}(n+p)$ en considérant en plus la validation du motif dans le cas où il est présent.

Ici, $p \leq n$, donc $\mathcal{O}(n+p) = \mathcal{O}(n)$, mais c'est important de garder en tête cette complexité en $\mathcal{O}(n+p)$ qu'on retrouvera car elle s'appliquera à des algorithmes où l'on effectue un **pré-traitement** sur le motif pour l'appliquer ensuite sur plusieurs chaînes.

Algorithme de Boyer-Moore

Dans un premier temps, on va présenter la variante usuelle de cet algorithme, appelée algorithme de Boyer-Moore-Horspool.

On présentera ensuite l'algorithme de Boyer-Moore en tant que tel.

Algorithme de Boyer-Moore-Horspool

Principe

Le principe de l'algorithme de Boyer-Moore-Horspool est d'effectuer une recherche du motif comme précédemment, mais en partant de la fin. On va alors tenter de trouver des suffixes de m de plus en plus grand.

- Si on trouve ainsi le motif m, on renvoie la position.
- Sinon, c'est qu'on a lu dans s un mot de la forme xm' où $x \in \Sigma$ et m' est un suffixe strict de m mais pas xm'.
  - Si x n'apparaît pas dans m, on peut alors relancer la recherche juste après x dans s.
  - Si x est présent dans m, on peut relancer la recherche en alignant ce caractère avec sa position la plus à droite dans m.

Algorithme de Boyer-Moore-Horspool

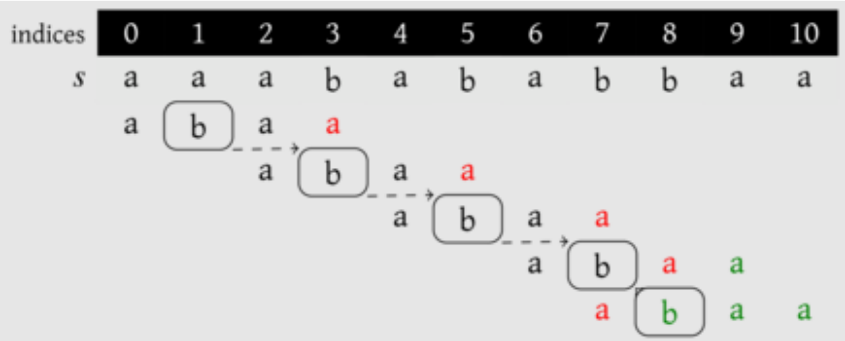
Remarque

Il faut tenir compte différemment du dernier caractère du motif, car il n'est pas utile de le réaligner.

On considère alors, quand elle existe, l'occurrence précédente de ce caractère.

On obtient ainsi une stratégie de saut qui en cas d'échec relance la recherche plus loin.

Algorithme de Boyer-Moore-Horspool



Exemple

Voici un premier exemple où l'on effectue la recherche de $abaa$ dans le mot $aaababababbaa$.

Lors des étapes avec une flèche en pointillés, on saute de plus d'un cran dans s pour aligner le dernier b de m avec le b trouvé dans s, ce qui fait gagner du temps.

Implémentation par table de sauts

Table de saut

Pour réaliser ces sauts, on construit une table `droite` indexée par Σ, et telle que `droite[c]` indique l'indice de l'occurrence la plus à droite dans le motif m du caractère c, en ignorant le dernier caractère du motif.

Exemple

Dans l'exemple précédent où $m = abaa$, on obtient :

c	'a'	'b'	'c'	...
`droite[c]`	2	1	$\emptyset$	...

Implémentation par table de sauts

Implémentation

Cette table contient donc de l'ordre de $|\Sigma|$ éléments.

On peut l'implémenter par un tableau de taille $|\Sigma|$, étant donné une numérotation des éléments de Σ.

On peut aussi l'implémenter par un dictionnaire, ce qui est plus économe en espace si le motif contient peu de lettres différentes.

On a choisi ici, pour des raisons pédagogiques, de considérer la numérotation ASCII naturelle associée aux caractères.

Implémentation par table de sauts

OCaml

```ocaml
let taille_alphabet = 256 ;;

let calcule_droite motif =
  let droite = Array.make taille_alphabet None in
  let p = String.length motif in
  for i = 0 to p-2 do
    let j = p-2-i in
    let c = motif.[j] in
    if droite.(Char.code c) = None
    then droite.(Char.code c) <- Some j
  done;
  droite
;;
```

Implémentation par table de sauts

C

```c
#include <string.h>
#include <stdlib.h>

int taille_alphabet = 256;

/* Calcule le tableau droite associé au motif
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_droite(char *motif)
{
  int *droite = (int*)malloc(sizeof(int) * taille_alphabet);
  memset(droite, -1, sizeof(int) * taille_alphabet); // Initialise le tableau avec des -1

  int p = strlen(motif);
  for (int i = 0; i < p-2; i++)
  {
    int j = p-2-i;
    char c = motif[j];
    if (droite[c] < 0)
      droite[c] = j;
  }
  return droite;
}
```

Destination du saut

Destination du saut

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer.

Si à la position $i + j$, on a un échec après avoir lu le caractère c où `droite[c]` contient la valeur k.

- Si $k = \emptyset$, c'est que le motif ne pourra jamais être trouvé tant que le caractère c sera présent : on relance donc la recherche juste après, à l'indice $i + j + 1$.

Destination du saut

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer.

Si à la position $i + j$, on a un échec après avoir lu le caractère c où `droite[c]` contient la valeur k.

- Si $k \geq j$, cela signifie que c est présent plus à droite dans le motif, donc aligner cette occurrence ne permettrait pas d'avancer la recherche. Rien ne nous permet de savoir si c apparaît ou non ailleurs dans le motif, on relance alors prudemment la recherche en $i + 1$.

Destination du saut

Destination du saut

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer.

Si à la position $i + j$, on a un échec après avoir lu le caractère c où `droite[c]` contient la valeur k.

- Si $k < j$, on veut aligner ce c avec le caractère correspondant du motif. Si on relance à l'indice i', on veut avoir $i' + k = i + j$, donc $i' = i + j - k$.

Algorithme de Boyer-Moore-Horspool

OCaml

```ocaml
exception Difference ;;
exception Trouve of int ;;

let recherche_BMH motif droite chaine =
  let n = String.length chaine in
  let p = String.length motif in
  let i = ref 0 in
  try
    while !i <= n-p do
      try
        for j = p-1 downto 0 do
          if chaine.[!i+j] <> motif.[j]
          then begin
            let dec = match droite.(Char.code chaine.[!i+j]) with
              | None -> j+1
              | Some k when k < j -> j-k
              | _ -> 1 in
            i := !i + dec;
            raise Difference
          end
        done;
        raise (Trouve !i)
      with Difference -> ()
    done;
    None
  with Trouve k -> Some k
;;
```

Algorithme de Boyer-Moore-Horspool

C

```c
int recherche_BMH(char *motif, int *droite, char *chaine)
{
  int n = strlen(chaine);
  int p = strlen(motif);
  int i = 0;
  while (i <= n-p)
  {
    bool present = true;
    for (int j = p-1; j >= 0; j--)
    {
      if (chaine[i+j] != motif[j])
      {
        int k = droite[chaine[i+j]];
        present = false;
        if (k < 0)
          i = i + j + 1;
        else if (k < j)
          i = i + j - k;
        else
          i = i + 1;
        break;
      }
    }
    if (present)
      return i;
  }
  return -1;
}
```

Algorithme de Boyer-Moore-Horspool

Terminaison

L'algorithme termine car le nouvel indice auquel on relance la recherche est toujours strictement plus grand que le précédent.

Algorithme de Boyer-Moore-Horspool

Correction

Il suffit de s'assurer que les indices écartés correspondent nécessairement à des recherches infructueuses.

Sans perte de généralité, on peut supposer que la recherche s'effectue depuis le premier indice de s.

Comme seul les sauts d'au moins deux indices sont ceux pour lesquels il est nécessaire de faire un preuve, cela correspond au cas où $m = m_1 c m_2 d m_3 x$ et $s = s_1 c m_3 s'$, avec $c, d, x \in \Sigma$, $d \neq c$, et c non présent dans $m2dm_3$.

Ainsi, toute recherche démarrant à des indices inférieurs échouera systématiquement, au plus tard, en comparant le caractère c de cm_3 avec un caractère du motif dans $m_2 d m_3$, donc différent de c.

Algorithme de Boyer-Moore-Horspool

Complexité

Tout d'abord, on remarque que la **table de saut** se construit en $\mathcal{O}(\max(|m|, |\Sigma|))$ pour un motif m sur l'alphabet Σ.

Sans chercher à rentrer dans les détails, on peut raisonnablement penser si l'alphabet contient assez de caractères que les motifs auront peu de répétitions et qu'ainsi les sauts seront presque toujours maximaux, ce qui permet d'obtenir de l'ordre de $\frac{n}{p}$ comparaisons où $n = |s|$ et $p = |m|$.

Algorithme de Boyer-Moore-Horspool

Complexité

Cependant, dans le pire des cas, cet algorithme n'est pas meilleur que le précédent.

La complexité dans le pire des cas de l'algorithme de Boyer-Moore-Horspool est donc en $\mathcal{O}(np)$, même si en pratique elle est sous-linéaire.

Exemple

Avec $s = a^n$ et $m = ba^{p-1}$, lors de la recherche à l'indice i il est nécessaire d'attendre de comparer b pour constater un échec et relancer la recherche en $i + 1$.

On va donc faire $\mathcal{O}(np)$ comparaisons.

Algorithme de Boyer-Moore-Horspool

Remarque

Si l'alphabet contient peu de caractères, ce qui est le cas en particulier du binaire ou de l'ADN, il y a de grandes chances pour qu'on soit dans le cas pire.

Ainsi, l'algorithme de Boyer-Moore-Horspool n'est pas adapté pour ce type de texte.

Algorithme de Boyer-Moore

Exemple : Principe de l'algorithme de Boyer-Moore

Considérons le cas ci-dessus de l'algorithme précédent : on cherche $m = abbcabc$ dans $l = cbacbbcabc$.

On remarque qu'en raison du fonctionnement de l'algorithme, on est forcé de faire de tous petits sauts, et on est ramené à l'algorithme naïf.

Cependant, après la première étape, on sait qu'on a lu un suffixe du motif bc, qui est précédé d'un caractère a : bbc n'est pas un suffixe de m.

Algorithme de Boyer-Moore

Exemple : Principe de l'algorithme de Boyer-Moore

Il y a un autre endroit dans le motif où on peut trouver $*bc$ avec $*$ un autre caractère que a.

On pourrait donc relancer la recherche en alignant cette occurrence de bc avec celle qu'on vient de lire.

Cela revient à sauter directement à la dernière étape dans cet exemple.

Algorithme de Boyer-Moore

Principe

Pour pouvoir réaliser ce décalage, il est nécessaire de calculer une nouvelle table en parcourant le motif pour identifier de telles apparitions de suffixes.

On peut aller plus loin en considérant également le plus long préfixe du motif qui soit un suffixe du suffixe considéré.

Notations et définitions

Langages

On réintroduira toutes les définitions qui vont suivre l'an prochain lors du chapitre sur les langages.

Dans ce contexte, on parle plutôt de **mot** que de chaîne de caractères.

Définition

Un **mot** sur l'**alphabet** Σ est donc une suite **finie** $a_1 \dots a_n$ de **lettres** dans l'alphabet.

On note ε l'unique **mot vide**, i.e. ne contenant aucune lettre.

L'**ensemble des mots** sur Σ est noté Σ^*.

Si u et v sont des mots, on note uv le mot obtenu par **concaténation**.

Notations et définitions

Remarque

Σ^* muni de la concaténation comme loi de composition interne possède une structure de **monoïde** :

- la loi est associative : $\forall u, v, w \in \Sigma^*,\ (uv)w = u(vw)$;
- ε est l'élément neutre de la loi : $\forall u \in \Sigma^*\ \varepsilon u = u\varepsilon = u$.

Cette structure très simple est cruciale en informatique.

Notations et définitions

Définition (préfixe, suffixe)

Soit $u, v \in \Sigma^*$. On dit que :

- v est un **préfixe** de u si $\exists w \in \Sigma^*$ tel que $u = vw$;
- v est un **suffixe** de u si $\exists w \in \Sigma^*$ tel que $u = wv$;

Si $w \neq \varepsilon$, on parle de préfixe ou suffixe **propre**.

On dit que v est un **bord** de u lorsque v est préfixe et suffixe propre de u.

Exemple

Soit $u = abacaba$.

$abac$ est un **préfixe**, $caba$ est un **suffixe**, et aba est un **bord**.

Notations et définitions

Définition (suffixes disjoints)

Soit $x = x_1 \dots x_n$ et u, v deux suffixes **distincts** de x.
On dit que u et v sont des **suffixes disjoints** quand on est dans l'un des cas suivants :

- $u = x$;
- $v = x$;
- $u \neq x$, $v \neq x$, et $x_{|x|-|u|} \neq x_{|x|-|v|}$.

Deux suffixes sont donc disjoints s'ils sont précédés dans x par des lettres différentes.

On définit de même la notion de **préfixes disjoints**.

Table des bons suffixes

Principe

On considère un motif $x = x_0 \dots x_{n-1}$.

On va construire une table `bonsuffixe` appelée **table des bons suffixes** du motif x et telle que, pour $i \in [\![0, n-1]\!]$, `bonsuffixe[i]` donne le nombre de positions dont on doit décaler le motif vers la droite pour relancer la recherche après la lecture du suffixe $x_{i+1} \dots x_{n-1}$.

Table des bons suffixes

Principe

Supposons que l'on vient de lire avec succès un suffixe propre u de x.

Ainsi, $x = x_0 \dots x_i u$, et on vient de lire dans s au avec $a \neq x_i$.

- Soit il existe un autre suffixe de x de la forme buv, où $b \neq x_i$, et alors on appelle bon suffixe pour u un tel suffixe de longueur minimale, et on pose `bonsuffixe[i]` $= |v|$.
- Sinon, on cherche v de longueur minimale tel que x soit un suffixe de uv, et on pose également `bonsuffixe[i]` $= |v|$.

On remarque que si x est suffixe de uv et qu'on a également buv' suffixe de x, alors $|uv| = |u| + |v| \geq |x| \geq |buv'| \geq |u| + |v'|$, donc $|v| \geq |v'|$, ce qui permet de considérer le plus petit v sur l'ensemble des cas.

Table des suffixes

Table des suffixes

Afin de calculer efficacement `bonsuffixe`, on va commencer par calculer la **table des suffixes** du motif : il s'agit de la table `suffixe` où `suffixe[i]` contient la longueur du plus long suffixe de x de la forme $x_j \dots x_i$.

Ainsi, si on note S_i les suffixes de cette forme, on a :

$$\texttt{suffixe[i]} = \begin{cases} 0 & \text{si } S_i = \emptyset \\ \max\{|s| \mid s \in S_i\} & \text{sinon} \end{cases}$$

Nécessairement, `suffixe[n-1]` $= n$, car x convient.

Table des suffixes

Exemple

Pour $x = bcabc$, on a :

i	0	1	2	3	4
`suffixe[i]`	0	2	0	0	5

Pour $x = abbabba$, on a :

i	0	1	2	3	4	5	6
`suffixe[i]`	1	0	0	4	0	0	7

Table des suffixes

Calcul efficace

Il est possible de construire `suffixe` avec un simple parcours linéaire en tirant partie de l'information déjà calculée. Pour cela, on va remplir `suffixe` de droite à gauche.

À tout moment, on va conserver le meilleur suffixe rencontré, i.e. celui pour lequel on est allé le plus loin à gauche avant d'avoir un échec de comparaison.

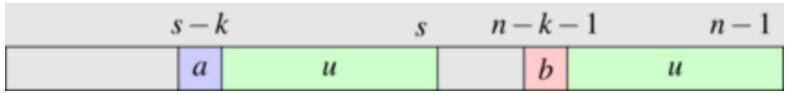
Table des suffixes

Calcul efficace

On note s la position la plus à droite de ce suffixe, et k sa longueur, il s'agit donc de $u = x_{s-k+1} \dots x_s$, et il y a eu un échec de comparaison en x_{s-k}.

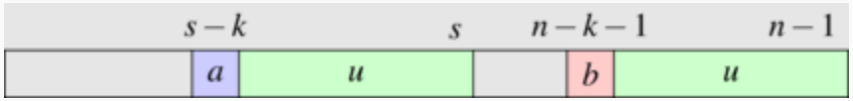
Par définition de `suffixe`, on a `suffixe[s]` $= k$.

Le mot x s'écrit alors :



Maintenant, on considère une position $s-i$ telle que $s > s-i > s-k$, cela signifie qu'on cherche un suffixe depuis une position interne au mot u de gauche.

Table des suffixes



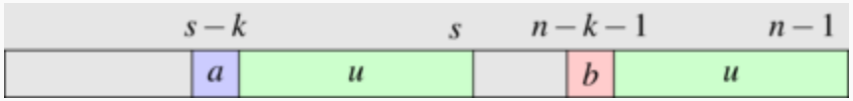
Calcul efficace

Le point clé permettant d'obtenir un algorithme linéaire est de remarquer que la situation est la même que dans le mot u de droite.

Or, comme on procède de gauche à droite, on a déjà calculé la valeur correspondante `suffixe[n-1-i]`.

Là, on a deux cas.

Table des suffixes



Calcul efficace

- Soit quand on a cherché le plus grand suffixe à partir de $n-1-i$, on s'est heurté à une erreur de comparaison en $n-1-k$. Dans ce cas, on a `suffixe[n-1-i]` $= k-i$, et on peut regarder, en partant de $s-k$, si on peut prolonger le suffixe finissant à la position $s-i$.

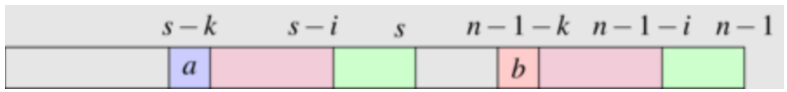
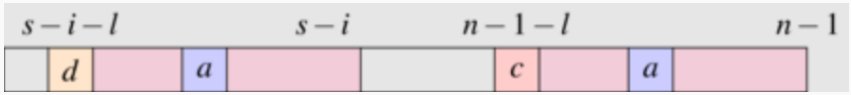


Table des suffixes

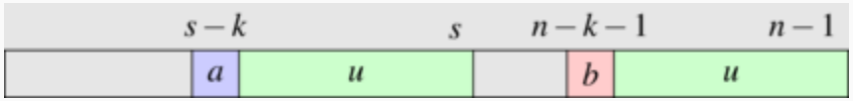


Calcul efficace

Pour effectuer ce prolongement, il suffit de comparer, caractère par caractère, vers la gauche en partant de la position $s - k$. On aboutira alors à une nouvelle position du suffixe finissant le plus à gauche qui finira en $s - i$.

Remarquons qu'il n'est pas nécessaire que $s - i - l \neq s - k$. C'est-à-dire que même si a ne permet pas de prolonger le suffixe déduit de la position $n - i$, on considère tout de même que la nouvelle position de référence est $s - i$. On en déduit également la valeur `suffixe[s-i]` $= l$.

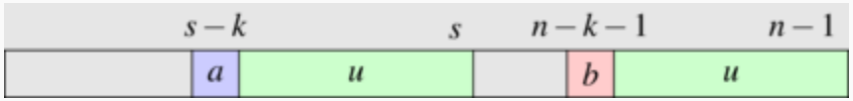
Table des suffixes



Calcul efficace

- Soit `suffixe[n-1-i]` $= p \neq k - i$, et alors :
  - soit $p < k - i$, on a alors pour ce suffixe un échec dans u, ce qui limite de la même manière la valeur en $s - i$: `suffixe[s-i]` $= p$;
  - soit $p > k - i$, donc on doit avoir un b après avoir le suffixe dans u depuis $s - i$ pour le prolonger, or c'est impossible car il y a un $a \neq b$. Ainsi, le suffixe est limité par u : `suffixe[s-i]` $= k - i$.

Table des suffixes



Calcul efficace

Il reste à traiter le cas où $s-i \leq s-k$, ce qui revient à considérer qu'on a dépassé le précédent suffixe pouvant apporter une information.

On procède donc naïvement pour trouver le plus grand suffixe depuis cette position.

Table des suffixes

OCaml

```ocaml
let calcule_suffixe x =
  let n = String.length x in
  let suffixe = Array.make n (-1) in
  suffixe.(n-1) <- n;
  let plus_a_gauche = ref (n-1) in
  let depart = ref (-1) in
  for j = n-2 downto 0 do
    if !plus_a_gauche < j
    && suffixe.(n-1- !depart+j) <> j - !plus_a_gauche
    then suffixe.(j) <- min suffixe.(n-1- !depart+j) (j - !plus_a_gauche)
    else begin
      plus_a_gauche := min j !plus_a_gauche;
      depart := j;
      while !plus_a_gauche >= 0
            && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_gauche] do
        plus_a_gauche := !plus_a_gauche - 1
      done;
      suffixe.(j) <- !depart - !plus_a_gauche
    end
  done;
  suffixe
;;
```

Table des suffixes

C

```c
int *calcule_suffixe(char *x)
{
  int n = strlen(x);
  int *suffixe = malloc(sizeof(int) * n);
  memset(suffixe, -1, sizeof(int) * n);
  suffixe[n-1] = n;

  int plus_a_gauche = n-1;
  int depart = -1;
  for (int j = n-2; j >= 0; j--)
  {
    if (plus_a_gauche < j
        && suffixe[n-1-depart+j] != j-plus_a_gauche)
      suffixe[j] = MIN(suffixe[n-1-depart+j], j-plus_a_gauche);
    else {
      plus_a_gauche = MIN(plus_a_gauche, j);
      depart = j;
      while (plus_a_gauche >=0
          && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
        plus_a_gauche--;
      suffixe[j] = depart - plus_a_gauche;
    }
  }
  return suffixe;
}
```

Table des suffixes

Terminaison et complexité

On remarque que dans ce code, `plus_a_gauche` ne peut que diminuer, on effectue donc au plus n itérations dans la boucle **`while`** pour tout l'algorithme.

Donc, en considérant la boucle **`for`**, on effectue au plus $2n$ comparaisons de caractères : au plus une pour chaque itération de la boucle **`for`** pour voir si on entre dans le **`while`**, puis en tout au plus n avant de sortir du **`while`**.

L'algorithme termine donc bien en temps $\mathcal{O}(|x|)$.

Obtention de `bonsuffixe` à partir de `suffixe`

Calcul de `bonsuffixe`

On reprend maintenant le calcul de `bonsuffixe[i]` dans le mot $x = x_0 \dots x_{n-1}$.

On cherche à obtenir des suffixes de x de la forme buv où $b \neq x_i$ et $u = x_{i+1} \dots x_{n-1}$ est un suffixe de x.

Mais si `suffixe[k-1]` $= n - 1 - i$, cela signifie que ce suffixe est exactement u, et qu'il est soit préfixe, soit précédé d'une lettre différente de x_i (sinon $n - 1 - i$ ne serait pas maximal).

Obtention de bonsuffixe à partir de suffixe

Calcul de bonsuffixe

On a donc :

$$\begin{aligned}\texttt{bonsuffixe[n-1-i]} &= \min\{n-1-k \mid \texttt{suffixe[k]} = n-1-i\} \\ &= n-1-\max\{k \mid \texttt{suffixe[k]} = n-1-i\}\end{aligned}$$

On remarque alors qu'on peut faire croitre k et poser :

$$\texttt{bonsuffixe[n - 1 - suffixe[k]]} = n-1-k$$

On aura alors naturellement, à la fin de la boucle, la valeur minimale placée en dernier.

Obtention de `bonsuffixe` à partir de `suffixe`

Calcul de `bonsuffixe`

Reste à considérer les valeurs non remplies ainsi dans le tableau `bonsuffixe`.

Elles correspondent aux positions i telles qu'il n'existe pas de suffixe de la forme buv.

On doit donc chercher un mot uv de longueur minimale dont x est suffixe. Mais u étant un suffixe de x, cela revient à considérer les **bords** de x.

La table `suffixe` permet également de détecter les bords : si $x_0 \dots x_k$ est un bord, c'est que `suffixe[k]` $= k + 1$.

Obtention de `bonsuffixe` à partir de `suffixe`

Calcul de `bonsuffixe`

Soit $k < n-1$ maximal vérifiant cette condition.

Pour tout $u = x_{i+1} \dots x_{n-1}$ suffixe de x, pour qu'il ait $x_0 \dots x_k$ comme suffixe, il faut qu'il soit strictement plus long (sinon on est dans le cas précédent), donc que $n - i > k + 1$, i.e. $i < n - 1 - k$.

Dans ce cas, x est alors suffixe de uv où $v = x_{k+1} \dots x_{n-1}$ donc $|v| = n - 1 - k$.

Les k plus petits ne pourront alors que faire augmenter $|v|$, on peut ainsi poser `bonsuffixe[i]` $= n - 1 - k$.

Obtention de `bonsuffixe` à partir de `suffixe`

Calcul de `bonsuffixe`

On en déduit un remplissage en parcourant les k dans l'ordre décroissant de $n-2$ à 0, tout en maintenant l'indice i de la prochaine valeur à remplir dans `bonsuffixe`.

Dès qu'on détecte un bord, on place $n-1-k$ jusqu'à ce que $i \geq n-1-k$.

En sortie de boucle, il est possible que $i < n$ donc qu'il reste des valeurs à remplir. On remarque dans ce cas là que pour que x soir un suffixe de uv, il faut que $v = x$. On a donc pour ces valeurs restantes `bonsuffixe[i]` $= n$.

Obtention de `bonsuffixe` à partir de `suffixe`

Calcul de `bonsuffixe`

Comme ce second cas est toujours plus long que le premier quand les deux se produisent en i, on implémente successivement les remplissages de sorte à obtenir la valeur minimum.

Obtention de bonsuffixe à partir de suffixe

OCaml

```
let calcule_bonsuffixe x =
  let n = String.length x in
  let suffixe = calcule_suffixe x in
  let bonsuffixe = Array.make n n in
  let suivant = ref 0 in
  for k = n-2 downto 0 do
    if suffixe.(k) = k+1 (* c'est un bord *)
    then begin
      for i = !suivant to n-2-k do
        bonsuffixe.(i) <- n-1-k
      done;
      suivant := n-1-k
    end
  done;
  for k = 0 to n-2 do
    bonsuffixe.(n-1-suffixe.(k)) <- n-1-k
  done;
  bonsuffixe
;;
```

Obtention de bonsuffixe à partir de suffixe

C

```c
int *calcule_bonsuffixe(char *x)
{
  int n = strlen(x);
  int *suffixe = calcule_suffixe(x);
  int *bonsuffixe = malloc(sizeof(int) * n);
  memset(bonsuffixe, n, sizeof(int) * n);

  int suivant = 0;
  for (int k = n-2; k >= 0; k--)
  {
    if (suffixe[k] == k+1) // bord
    {
      for (int i = suivant; i < n-1-k; i++)
        bonsuffixe[i] = n-1-k;
      suivant = n-1-k;
    }
  }
  for (int k = 0; k < n-1; k++)
  {
    bonsuffixe[n-1-suffixe[k]] = n-1-k;
  }

  free(suffixe);

  return bonsuffixe;
}
```

Obtention de `bonsuffixe` à partir de `suffixe`

Complexité

Il est facile de constater que cet algorithme est de complexité $\mathcal{O}(|x|)$.

Algorithme de Boyer-Moore

On incorpore naturellement la table précédente à l'algorithme de Boyer-Moore, en choisissant le meilleur décalage entre cette table et la stratégie précédente.

Algorithme de Boyer-Moore

OCaml

```
let recherche_BM motif droite bonsuffixe chaine =
  let n = String.length chaine in
  let p = String.length motif in
  let i = ref 0 in
  try
    while !i <= n-p do
      try
        for j = p-1 downto 0 do
          if chaine.[!i+j] <> motif.[j]
          then begin
            let dec =  match droite.(Char.code chaine.[!i+j]) with
              | None -> j+1
              | Some k when k < j -> j-k
              | _ -> 1 in
            i := !i + max dec bonsuffixe.(j);
            raise Difference
          end
        done;
        raise (Trouve !i)
      with Difference -> ()
    done;
    None
  with Trouve k -> Some k
;;
```

Algorithme de Boyer-Moore

C

```c
int recherche_BM(char *motif, int *droite, int *bonsuffixe, char *chaine)
{
  int n = strlen(chaine);
  int p = strlen(motif);
  int i = 0;
  while (i <= n-p)
  {
    bool present = true;
    for (int j = p-1; j >= 0; j--)
    {
      if (chaine[i+j] != motif[j])
      {
        int k = droite[chaine[i+j]];
        int dec = 1;
        present = false;
        if (k < 0)
          dec = j + 1;
        else if (k < j)
          dec = j - k;
        i = i + MAX(dec, bonsuffixe[j]);
        break;
      }
    }
    if (present){ return i; }
  }
  return -1;
}
```

Algorithme de Boyer-Moore

Complexité

Supposons que le motif est de longueur p, que la chaîne dans laquelle on recherche est de longueur n et que la taille de l'alphabet est une constante indépendante des entrées.

La première partie de l'algorithme consiste à construire les tables de sauts, comme on l'a vu, elle est en complexité en temps et en espace en pire cas en $\mathcal{O}(p)$.

On admet que l'algorithme Boyer-Moore complet, étant donné les deux tables de saut et d'autres modifications mineures non présentées ici, est en complexité temporelle en pire cas en $\mathcal{O}(n)$.

Algorithme de Boyer-Moore

Complexité

Il est assez raisonnable de penser que soit $p \leq n$ quand on effectue une recherche, soit on compte chercher un même motif dans plusieurs textes et on réutilise ainsi les tables de sauts.

Il n'est donc pas forcément très pertinent de parler de la complexité globale de l'algorithme, mais lorsqu'on le fait, on dit qu'elle est en $\mathcal{O}(n+p)$.

On rappelle ici le rôle de l'addition dans les complexités qui fait référence à la succession de deux traitements, un en $\mathcal{O}(p)$, suivi d'un en $\mathcal{O}(n)$.

Algorithme de Rabin-Karp

Principe

L'algorithme de **Rabin-Karp** est un algorithme de recherche d'un motif dans un texte qui utilise une notion d'**empreinte** pour déterminer, en temps constant, s'il est probable que la position actuelle corresponde à une occurrence du motif.

Algorithme de Rabin-Karp

Principe

Pour cela, si on cherche un motif de longueur p sur l'alphabet Σ, on considère une **fonction de hachage** $h : \Sigma^p \to X$.
Les éléments de l'ensemble X sont appelés des empreintes, et on suppose que l'égalité entre deux empreintes se vérifie en temps constant, contrairement à l'égalité dans Σ^p qui se vérifie en $\mathcal{O}(p)$ dans le pire des cas.

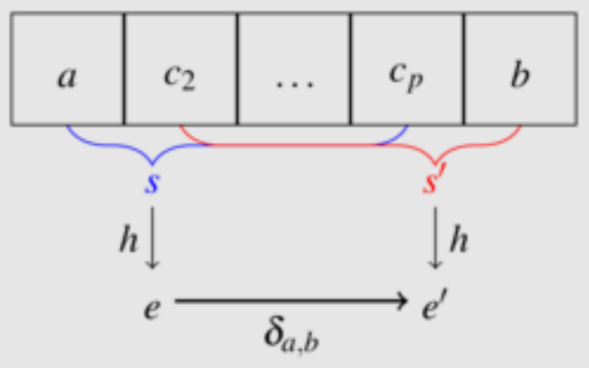
Le plus souvent, on choisit pour X le type `int`.

Algorithme de Rabin-Karp

Principe

Bien qu'il soit normalement aussi coûteux de calculer l'image par h d'une sous-chaîne de longueur p que de tester l'égalité entre cette sous-chaîne et le motif, le point essentiel de l'algorithme de Rabin-Karp est d'utiliser une fonction de hachage permettant un **calcul incrémental en temps constant**.

Algorithme de Rabin-Karp



Principe

Ici, on considère donc, pour $a, b \in \Sigma$, une fonction de mise à jour $\delta_{a,b} : X \to X$ telle que :

$$\forall c_2, \ldots, c_p \in \Sigma,\ \delta_{a,b}(h(ac_2 \ldots c_p)) = h(c_2 \ldots c_p b)$$

Algorithme de Rabin-Karp

Algorithme 1 : Algorithme de Rabin-Karp

Données : une chaîne $s = c_0 \dots c_{n-1}$, un motif m

Résultat : un indice i de s où le motif m apparaît (ou -1 si un tel i n'existe pas)

$n \leftarrow longueur(s)$;

$p \leftarrow longueur(m)$;

$e_m \leftarrow h(m)$;

$e \leftarrow h(c_0 \dots c_{p-1})$;

pour $i \in [\![0, n-p]\!]$ **faire**

    si $e_m = e$ **alors**

        si $m = c_i \dots c_{i+p-1}$ **alors**

            retourner i ;

    si $i < n - p$ **alors**

        $e \leftarrow \delta_{c_i, c_{i+p}}(e)$;

retourner -1 ;

Algorithme de Rabin-Karp

Complexité

La complexité temporelle liée à la gestion des empreintes est donc en $\mathcal{O}(n+p) = \mathcal{O}(n)$ car $n \geq p$.

Par contre, pour calculer la complexité liée aux tests $m = c_i \dots c_{i+p-1}$, il est nécessaire d'estimer la proportion de **faux positifs**, i.e. de positions i telles que $e_m = e$ mais $m \neq c_i \dots c_{i+p-1}$.

On va voir dans la partie suivante qu'on peut supposer qu'elle est négligeable, ce qui permet de considérer que l'algorithme de Rabin-Karp est **linéaire**.

Choix de la fonction de hachage

Fonction de hachage

Réaliser une bonne fonction de hachage est une question très complexe qui dépasse le cadre du cours de MP2I. Cependant, il est possible de réaliser ici une fonction de hachage répondant aux contraintes de Rabin-Karp assez facilement.

Pour cela, on considère que les caractères sont des entiers compris entre 0 et 255, ce qui correspond au type **`char`**. On peut alors identifier une chaîne de longueur p avec un nombre entre 0 et $r^p - 1$ où $r = 2^8$:

$$P(c_0 \dots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \dots + c_{p-1}$$

Choix de la fonction de hachage

$$P(c_0 \dots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \cdots + c_{p-1}$$

Fonction de hachage

On considère de plus un entier premier q, et on pose :

$$h(s) = P(s) \mod q$$

On peut alors définir :

$$\delta_{a,b}(e) = (r(e - ar^{p-1}) + b) \mod q$$

Si on précalcule $r^{p-1} \mod q$, il suffit d'un nombre d'opération constant, et indépendant de p, pour calculer la nouvelle empreinte à l'aide de $\delta_{a,b}$.

Choix de la fonction de hachage

Choix du nombre premier

Le point essentiel est alors de déterminer un nombre premier q tel qu'il soit peu probable d'obtenir des faux positifs.

Une analyse mathématique permet d'affirmer que chaque élément de $[\![0, q-1]\!]$ a de l'ordre de $\frac{r^p}{q}$ antécédents par h.

Ainsi, si on choisit deux chaînes aléatoirement dans Σ^p, il y aura collision avec probabilité proche de $\frac{1}{q}$.

En considérant q proche de la taille maximale pour le type entier considéré, on minimise donc cette probabilité.

Choix de la fonction de hachage

Exemple : Choix du nombre premier

Un exemple classique est $q = 2^{31} - 1$, car en plus d'être un nombre premier proche de `max_int`, on peut déduire $a \mod q$ à partir de l'écriture de a en base 2^{31}, et l'implémenter très efficacement à l'aide d'opérations bit à bit.

Algorithme de Rabin-Karp

OCaml

```ocaml
let hash r q s =
  let p = ref 1 in
  let e = ref 0 in
  for i = String.length s - 1 downto 0 do
    e := (!p * (Char.code s.[i]) + !e) mod q;
    p := (r * !p) mod q
  done;
  !e
;;

let delta r q rp a b e = (* rp est r^(p-1) mod q *)
  (r * (e - rp * (Char.code a)) + Char.code b) mod q
;;
```

Algorithme de Rabin-Karp

OCaml

```
exception Trouve of int ;;

let rabin_karp m s =
  let n = String.length s in
  let p = String.length m in
  let r = 256 in
  let q = 0x7fffffff in (* 2^(31)-1 *)
  let rp = pow r (p-1) q in
  let me = hash r q m in
  let e = ref (hash r q (String.sub s 0 p)) in
  try
    for i = 0 to n-p+1 do
      if me = !e && m = String.sub s i p then raise (Trouve i);
      if i+p < n then e := delta r q rp s.[i] s.[i+p] !e
    done;
    None
  with Trouve k -> Some k
;;
```

Algorithme de Rabin-Karp

C

```c
int hash(int r, int q, char *s, int n)
{
  int p = 1;
  int e = 0;
  for (int i = n-1; i >= 0; i--)
  {
    e = (p * s[i] + e) % q;
    p = (r * p) % q;
  }
  return e;
}

int delta(int r, int q, int rp, char a, char b, int e)
{
  return (r * (e - rp * a) + b) % q;
}
```

Algorithme de Rabin-Karp

C

```c
#include <string.h>

int rabin_karp(char *m, char *s)
{
  const int r = 256;
  const int q = 0x7fffffff;
  const int p = strlen(m);
  const int n = strlen(s);
  const int rp = powmod(r,p-1,q);
  const int me = hash(r,q,m,p);
  int e = hash(r,q,s,p);
  for (int i=0; i <n-p+1; i++)
  {
    if (me == e && strncmp(m,(s+i),p) == 0)
      return i;
    if (i+p < n)
      e = delta(r,q,rp,s[i],s[i+p],e);
  }
  return -1;
}
```

Algorithme de Rabin-Karp

Remarque

On se sert ici de l'évaluation paresseuse du && pour n'effectuer le test coûteux d'égalité des chaînes qu'en cas d'égalité des empreintes.

Algorithme de Rabin-Karp

Remarque

Su l'on suppose qu'il est improbable d'obtenir un faux positif, il est possible de renvoyer un succès dès que les empreintes sont égales.

L'avantage d'une telle version est alors d'être un algorithme sans retour sur les données, i.e. qu'il n'est pas nécessaire de garder en mémoire ou de réaccéder à un caractère.

Compression

Compression

Principe

On s'intéresse ici à la **compression** parfaite d'un texte, c'est-à-dire, étant donné un alphabet Σ, qu'on cherche à implémenter deux fonctions $comp$, $dec : \Sigma^* \to \Sigma^*$ telles que :

- $\forall m \in \Sigma^*,\ dec(comp(m)) = m$;
- pour la plupart des mots m qui correspondent aux données qu'on cherche à compresser : $|comp(m)| < |m|$.

Compression

Remarque

$dec \circ comp = id_{\Sigma^*}$ implique que $comp$ est **injective**.

Si A et B sont deux ensembles finis tels que $|A| > |B|$, il n'existe pas de fonction injective de A dans B. Ainsi, si on note L_n les mots de Σ^* de longueur au plus n, il ne peut exister de fonction injective de L_m dans L_n avec $m > n$.

Autrement dit, il est impossible d'espérer pouvoir compresser toutes les données de L_m. Si certains mots vont diminuer en longueur après compression, d'autres vont nécessairement augmenter.

Tout l'enjeu des algorithmes de compression parfaite est alors de diminuer les longueurs des mots qui nous intéressent.

Compression

Exemple

Si on s'intéresse à des mots issus de textes en français, il est plus important d'arriver à compresser une phrase usuelle comme "**`ceci est un texte`**" plutôt qu'une chaîne de caractères comme "**`c2#$%1ajdn //@#3d!fn`**".

Algorithme de Huffman

Principe

On va étudier ici un principe de **compression parfaite** appelé l'**algorithme de Huffman**, qui repose sur ce principe simple : coder sur moins de bits les caractères les plus fréquents.

Algorithme de Huffman

Exemple

Si on considère $m = abaabc$, en le codant avec un nombre de bits fixes, par exemple 2 avec le code $a = 00$, $b = 01$, $c = 10$, on aurait besoin de 12 bits pour représenter le mot.

Mais si on choisit le code suivant : $a = 0$, $b = 10$, $c = 11$, il suffit de 9 bits.

On a donc gagné 3 bits, soit un **facteur de compression** de 75%.

Algorithme de Huffman

Remarque

Dans l'exemple précédent, on remarque que pour pouvoir décompresser, il n'aurait pas été possible de faire commencer le code de b ou c par un 0, sinon on aurait eu ambigüité avec la lecture d'un a.

On parle alors de **code préfixe**.

Algorithme de Huffman

Définition (code préfixe)

Soit $X \subset \{0,1\}^*$. On dit que X est un **code préfixe** lorsque pour tous $x, y \in X$, x n'est pas un préfixe de y, et y n'est pas un préfixe de x.

Algorithme de Huffman

Code préfixe optimal

On se pose alors la question du **code préfixe optimal** pour un texte donné.

Soit Σ un alphabet fini et $f : \Sigma \to \mathbb{N}$ l'application associant à chaque lettre son nombre d'occurrences dans le texte considéré. Ainsi, $\sum_{x \in \Sigma} f(x)$ est la longueur du texte. On cherche un code préfixe X et une application $c : \Sigma \to X$ telle que $\sum_{x \in \Sigma} f(x)|c(x)|$ soit **minimale**, car cela correspond au nombre de bits après codage.

Algorithme de Huffman

Remarque

On utilise aussi la notion de **fréquence** d'une lettre, qui est son nombre d'occurrence rapporté à la longueur du texte.

Un des avantages de la notion de fréquence est qu'il est possible de considérer une **table de fréquence** déjà construite comme celle de la langue française.

Algorithme de Huffman

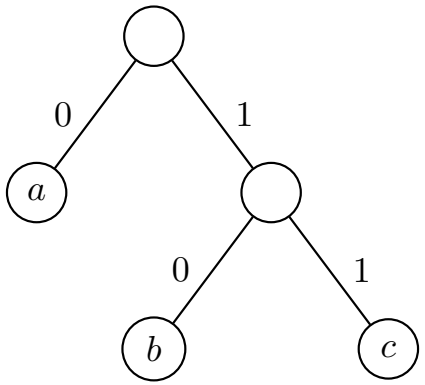
Arbre de Huffman

L'application de codage c peut être représentée par un **arbre binaire** où les arêtes gauches correspondent à 0, les arêtes droites à 1, et les feuilles aux éléments de Σ dont les étiquettes des chemins y menant depuis la racine de l'arbre correspondent à leur image par c.

Algorithme de Huffman

Exemple : Arbre de Huffman

Pour le code $a = 0, b = 10, c = 11$ précédent, on aurait l'**arbre de Huffman** suivant :



Algorithme de Huffman

Arbre de Huffman

Avec un tel arbre, il est très simple de décoder le texte codé, car il suffit de :

- suivre un chemin dans l'arbre jusqu'à tomber sur un feuille ;
- produire la lettre correspondante ;
- repartir de la racine de l'arbre.

La longueur du code associé à un lettre est alors égale à la profondeur de la feuille correspondante.

Algorithme de Huffman

Arbre de Huffman

L'optimalité du codage préfixe est ainsi équivalente à la minimalité de l'arbre vis-à-vis de la fonction d'objectif :

$$\varphi(t) = \sum_{x \in \Sigma} f(x) p(t, x)$$

où $p(t, x)$ est la profondeur de la feuille d'étiquette x dans l'arbre t (ou 0 si x n'est pas une étiquette de t).

Cette extension permet d'étendre la fonction d'objectif aux solutions partielles.

Algorithme glouton et implémentation

Algorithme de Huffman

L'algorithme de Huffman va construire un arbre correspondant à un codage optimal à l'aide d'une **file de priorité** d'arbres.

On étend pour cela l'application f à de tels arbres en définissant que si t est un arbre de feuilles $x_1, \ldots, x_n$ alors :

$$f(t) = f(x_1) + \cdots + f(x_n)$$

Algorithme glouton et implémentation

Algorithme de Huffman

On construit l'**arbre de Huffman** de la manière suivante :

- Au départ, on place dans la file des arbres réduits à une feuille, pour chaque élément $x \in \Sigma$, et dont la priorité est $f(x)$.
- Tant que la file contient au moins 2 éléments :
  - on retire les deux plus petits éléments x et y de la file de priorité (de priorités respectives $f(x)$ et $f(y)$) ;
  - on ajoute un arbre $z = \texttt{N}(x, y)$ de priorité $f(z) = f(x) + f(y)$.
- On renvoie l'unique élément restant dans la file.

Algorithme glouton et implémentation

```
1  type 'a fp = { mutable n : int; tab : 'a array; }
2  val creer_fp : int -> 'a -> 'a fp = <fun>
3  val est_vide_fp : 'a fp -> bool = <fun>
4  val enfiler_fp : 'a fp -> 'a -> unit = <fun>
5  val supprimer_max_fp : 'a fp -> 'a = <fun>
```

File de priorité

On réutilise la structure de **tas-max** codée au chapitre 15.

Puisqu'on voudrait ici une file de priorité **min**, il suffit d'utiliser $-f(x)$ au lieu de $f(x)$ pour la priorité de chaque arbre.

Algorithme glouton et implémentation

Arbre de Huffman

```
1  type arbre_huffman = F of int | N of arbre_huffman * arbre_huffman ;;
2
3  let construit_arbre occ =
4    let arbres = creer_fp 256 (0, F 0) in
5    for i=0 to 255 do
6      let f = occ.(i) in
7      if f > 0 (* on ignore les occurrences nulles *)
8      then enfiler_fp arbres (-f, F i)
9    done ;
10   while arbres.n > 1 do
11     let fx, x = supprimer_max_fp arbres in
12     let fy, y = supprimer_max_fp arbres in
13     enfiler_fp arbres (fx+fy, N(x,y))
14   done ;
15   let _, t = supprimer_max_fp arbres in
16   t
17 ;;
```

Algorithme glouton et implémentation

Algorithme glouton

L'algorithme de Huffman est un algorithme **glouton** car si on considère pour solution partielle la forêt présente dans la file et pour objectif la fonction φ étendue aux forêts en sommant la valeur de φ sur chaque arbre, alors fusionner dans la forêt F deux arbres x et y en la transformant en une forêt F' va avoir l'impact suivant sur la fonction d'objectif :

$$\varphi(F') = \varphi(F) + f(x) + f(y)$$

Algorithme glouton et implémentation

Algorithme glouton

En effet, on va rajouter 1 à la profondeur de chaque feuille, et donc la contribution de x passe de $\varphi(x) = \sum_{c \in x} f(c)p(x,c)$ à

$$\sum_{c \in x} f(c)(p(x,c)+1) = \varphi(x) + \sum_{c \in x} f(c) = \varphi(x) + f(x)$$

On remarque ainsi que la fusion qui minimise localement φ est celle qui fusionne les deux arbres de plus petite valeur pour f.

Preuve d'optimalité

Optimalité

Pour montrer que l'algorithme glouton produit ici un codage minimal, on va utiliser une technique classique qui consiste à montrer qu'étant donné une solution optimale, on peut toujours la transformer sans augmenter sa valeur pour obtenir, de proche en proche, la solution renvoyée par le glouton.

Preuve d'optimalité

Théorème

Supposons que les lettres les moins fréquentes soient a et b. Alors il existe un arbre optimal dont les deux feuilles étiquetées par a et b descendent du même nœud et sont de profondeur maximale.

Preuve d'optimalité

Preuve

Considérons un arbre optimal t, et soit c l'étiquette d'une feuille de profondeur maximale. On remarque qu'elle a forcément une feuille sœur, car sinon on pourrait omettre le nœud et l'arbre obtenu serait de plus petite valeur par φ.

Soit d l'étiquette de cette feuille sœur. Sans perte de généralité, on suppose $f(c) \leq f(d)$ et $f(a) \leq f(b)$. Comme a a le plus petit nombre d'occurrences, on a $f(a) \leq f(c)$, et comme b est la deuxième, on a $f(b) \leq f(d)$.

De plus, $p(t, a) \leq p(t, c)$ et $p(t, b) \leq p(t, d)$.

Preuve d'optimalité

Preuve

Si on échange les étiquettes a et c dans t pour obtenir un arbre t', alors seuls les termes associés à ces lettres changent dans l'évaluation de φ. On a donc :

$$\begin{aligned}\varphi(t') &= \varphi(t) - f(a)p(t,a) - f(c)p(t,c) + f(a)p(t,c) + f(c)p(t,a) \\ &= \varphi(t) + (\underbrace{f(c) - f(a)}_{\geq 0})(\underbrace{p(t,a) - p(t,c)}_{\leq 0}) \\ &\leq \varphi(t)\end{aligned}$$

L'échange préserve le caractère optimal. En fait, ici, on a nécessairement une égalité pour ne pas aboutir à une contradiction, donc soit les feuilles étaient à même profondeur, soit les lettres avaient le même nombre d'occurrences.

Preuve d'optimalité

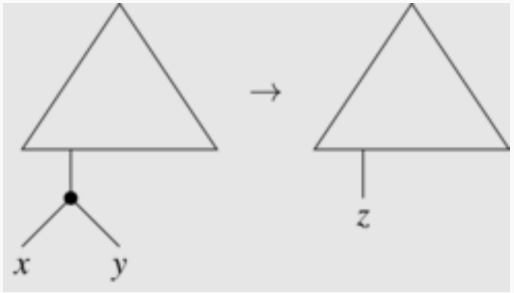
Preuve

Comme on a les mêmes relations entre b et d, on peut effectuer le même argument et échanger les étiquettes en préservant le caractère optimal.

Récurrence

Le théorème suivant va nous permettre de raisonner par récurrence en diminuant le nombre de lettres.

Preuve d'optimalité

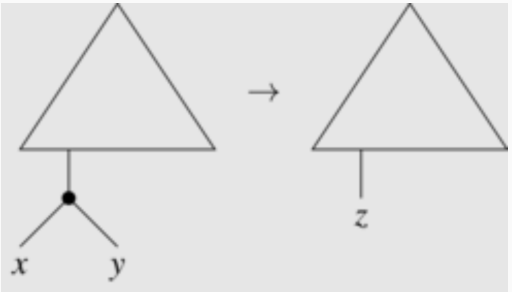


Théorème

Soit t un arbre ayant x et y comme feuilles sœurs, et t' l'arbre obtenu en remplaçant le nœud liant x et y par une feuille étiquetée par z, où z est une nouvelle lettre telle que $f(z) = f(x) + f(y)$.

On a alors $\varphi(t) = \varphi(t') + f(z)$.

Preuve d'optimalité



Preuve

Seuls les termes portant sur x, y, et z sont influencés par le changement, et on a :

$$\begin{aligned}\varphi(t) &= \varphi(t') + f(x)p(t,x) + f(y)p(t,y) - f(z)p(t',z)\\ &= \varphi(t') + f(z)(p(t',z)+1) - f(z)p(t',z)\\ &= \varphi(t') + f(z)\end{aligned}$$

Preuve d'optimalité

Théorème

L'algorithme de Huffman renvoie un arbre optimal.

Preuve

Par récurrence sur $|\Sigma|$.

- Si Σ ne contient qu'une seule lettre, il n'y a qu'un arbre qui est nécessairement optimal.
- Si la propriété est vraie pour un alphabet de $n - 1 \geq 1$ lettres, alors soit Σ contenant n lettres et x et y les deux lettres les moins fréquentes.
  On pose Σ' l'alphabet obtenu en remplaçant x et y par une nouvelle lettre z, et on suppose que $f(z) = f(x) + f(y)$.

Preuve d'optimalité

Preuve

L'hypothèse de récurrence assure qu'on obtient un arbre optimal t' en appliquant l'algorithme de Huffman sur Σ'.

Comme la première étape de l'algorithme va fusionner les feuilles x et y, on sait que l'arbre obtenu en partant de Σ se déduit de t' en remplaçant z par $\mathbf{N}(x, y)$.

Le théorème précédent assure alors que $\varphi(t) = \varphi(t') + f(z)$.

Preuve d'optimalité

Preuve

Soit t_o un arbre optimal pour Σ dans lequel x et y sont sœurs, possible en vertu du premier théorème, et soit t'_o l'arbre obtenu en remplaçant dans t_o le nœud liant x et y par une feuille étiquetée z.

On a ici encore $\varphi(t_o) = \varphi(t'_o) + f(z) \geq \varphi(t') + f(z) = \varphi(t)$ car t' est optimal.

Ainsi, on a bien l'égalité $\varphi(t_o) = \varphi(t)$, et t est optimal.

Gestion de fichiers en OCaml

OCaml

On va maintenant utiliser des fonctions OCaml permettant d'ouvrir/fermer un fichier, et de lire/écrire un octet dans un fichier :

- `open_in_bin : string -> in_channel` : prend un nom de fichier en argument, et renvoie un "input channel" pointant vers le début du fichier (**ouvre** le fichier en **lecture**).
- `input_byte : in_channel -> int` : renvoie le prochain **octet** du fichier, ou lève l'exception `End_of_file` si on est déjà à la fin du fichier.
- `close_in : in_channel -> unit` : **ferme** le fichier.

Gestion de fichiers en OCaml

OCaml

On va maintenant utiliser des fonctions OCaml permettant d'ouvrir/fermer un fichier, et de lire/écrire un octet dans un fichier :

- `open_out_bin : string -> out_channel` : prend un nom de fichier en argument, et renvoie un "output channel" pointant vers le fichier (**ouvre** le fichier en **écriture**).
- `output_byte : out_channel -> int -> unit` : écrit un octet dans le fichier (l'entier fourni est pris modulo 256).
- `close_out : out_channel -> unit` : **ferme** le fichier.

Calcul de la table d'occurrences

```
1  let table_occurrences nomdefichier =
2    let fichier = open_in_bin nomdefichier in
3    let occurrences = Array.make 256 0 in
4    begin
5      try
6        while true do
7          let c = input_byte fichier in
8          occurrences.(c) <- occurrences.(c) + 1;
9        done
10     with End_of_file -> ()
11   end;
12   close_in fichier;
13   occurrences
14 ;;
```

Table d'occurrence

On calcule une table d'occurrences pour l'ensemble des valeurs d'octets entre 0 et 255.

Sérialisation de l'arbre de Huffman

Sérialisation

Afin de décompresser, il est nécessaire de connaître l'arbre de Huffman donnant le code préfixe.

Pour cela, il faut sauvegarder cet arbre dans le fichier comme une série d'octets : on parle de **sérialisation**.

On choisit ici la représentation récursive `repr(a)` de l'arbre a définie ainsi :

- si `a = N(g,d)`, `repr(a) = 0 repr(g) repr(d)`;
- si `a = F(c)`, `repr(a) = 1 c`.

Sérialisation de l'arbre de Huffman

Remarque

Cela consiste à faire un parcours en profondeur **infixe** de l'arbre, avec comme traitement de chaque nœud :

- écrire `0` si c'est un nœud interne ;
- écrire `1` puis l'étiquette de la feuille si c'est un feuille.

Comme on l'a vu dans le chapitre sur les arbres, on peut reconstituer l'arbre sans ambiguïté à partir de son écriture infixe.

Exemple

Si `a = N(F 42, N(F 16, F 64))`, on obtient la suite d'octets :

```
0 1 42 0 1 16 1 64
```

Sérialisation de l'arbre de Huffman

```
let rec output_arbre f a = match a with
  | N(x, y) ->
    output_byte f 0;
    output_arbre f x;
    output_arbre f y
  | F c ->
    output_byte f 1;
    output_byte f c
;;

let rec input_arbre f =
  let code = input_byte f in
  match code with
  | 0 ->
    let g = input_arbre f in
    let d = input_arbre f in
    N(g,d)
  | _ -> F (input_byte f)
;;
```

Écriture dans un fichier un bit à la fois

Bit vs Octet

Le propre de l'algorithme de Huffman est d'associer à chaque caractère un codage binaire de longueur variable.

Le problème qu'il nous reste à régler est alors le suivant : dans un fichier, on ne peut pas lire/écrire bit à bit, mais seulement **octet par octet**.

Ainsi, afin de pouvoir écrire le codage de Huffman dans un fichier, il est nécessaire de regrouper les bits par **paquets de** 8.

Écriture dans un fichier un bit à la fois

char	a	b	c
code(char)	0	100	101

Exemple

Avec le codage ci-dessus, pour encoder le mot $abbaca$, on obtient le mot binaire `010010001010` qu'on complète avec des `0` à la fin, et qu'on sépare en octets : `01001000 10100000`.

On obtient donc les deux octets, convertis en décimal, 72 et 160.

Ce sont eux qu'on va écrire dans un fichier.

Écriture dans un fichier un bit à la fois

Principe

Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit, ainsi que le nombre de bits qui ont été accumulés.

Dès qu'on a accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis réinitialiser ces variables.

Quand on rajoute un bit b à l'accumulateur, on veut passer de $acc = b_1 \dots b_k$ à $b1 \dots b_k b = 2acc + b$.

Écriture dans un fichier un bit à la fois

```
type out_channel_bits = {
  o_fichier : out_channel;
  mutable o_accumulateur : int;
  mutable o_bits_accumules : int
};;

let open_out_bits fn = {
  o_fichier = open_out_bin fn;
  o_accumulateur = 0;
  o_bits_accumules = 0
};;

let output_bit f b =
  if f.o_bits_accumules = 8
  then begin
    output_byte f.o_fichier f.o_accumulateur;
    f.o_accumulateur <- 0;
    f.o_bits_accumules <- 0
  end;
  f.o_accumulateur <- 2 * f.o_accumulateur + (if b then 1 else 0);
  f.o_bits_accumules <- 1 + f.o_bits_accumules
;;
```

Écriture dans un fichier un bit à la fois

Padding

Il reste alors à traiter la question des **zéros** finaux : si l'accumulateur contient k bits au moment de la fermeture du fichier, où $0 < k < 8$, il faut ajouter $8 - k$ zéros.

On appelle cela du **padding** (qui veut dire **rembourrage** en anglais).

Ici, cela correspond à faire un **décalage vers la gauche** d'autant (fonction `Int.shift_left` en OCaml).

Comme il sera nécessaire de se souvenir que ces zéros ne sont pas significatifs à la lecture, on rajoute un octet final contenant cette valeur k.

Écriture dans un fichier un bit à la fois

```
let close_out_bits f =
  if f.o_bits_accumules = 0
  then output_byte f.o_fichier 0
  else begin
    let padding = 8 - f.o_bits_accumules in
    output_byte f.o_fichier (Int.shift_left f.o_accumulateur padding);
    output_byte f.o_fichier padding
  end;
  close_out f.o_fichier
;;
```

Écriture dans un fichier un bit à la fois

Lecture

Pour la lecture, on procède de même en faisant attention à deux points.

- On va lire les bits dans l'octet de la gauche vers la droite, c'est à dire du bit de poids le plus fort au bit de poids le plus faible. Ainsi, si l'accumulateur contient $acc = b_1 \dots b_8$, on peut obtenir b_1 en testant si $acc \geq 128$, et passer à $2 * acc \mod 256 = b_2 \dots b_8 0$.

Écriture dans un fichier un bit à la fois

Lecture

Pour la lecture, on procède de même en faisant attention à deux points.

- On doit tenir compte des zéros finaux : pour ça, on a besoin de savoir qu'on est en train de lire l'avant dernier caractère du fichier. On calcule donc la taille du fichier (avec `in_channel_length`), et on teste si l'octet lu est l'avant dernier (avec `pos_in`), auquel cas on lit le dernier octet, et on diminue d'autant le nombre de bits significatifs dans l'accumulateur.

Écriture dans un fichier un bit à la fois

```
1  type in_channel_bits = { i_fichier : in_channel; mutable i_accumulateur : int;
2                           mutable i_bits_accumules : int; i_taille : int };;
3
4  let open_in_bits fn =
5    let fichier = open_in_bin fn in
6    { i_fichier = fichier; i_accumulateur = 0;
7      i_bits_accumules = 0; i_taille = in_channel_length fichier }
8  ;;
9
10 let input_bit f =
11   if f.i_bits_accumules = 0
12   then begin
13     f.i_accumulateur <- input_byte f.i_fichier;
14     f.i_bits_accumules <- 8;
15     if pos_in f.i_fichier = f.i_taille - 1 (* dernière position *)
16     then
17       begin
18         let padding = input_byte f.i_fichier in
19         f.i_bits_accumules <- f.i_bits_accumules - padding
20       end
21   end;
22   let bit = f.i_accumulateur >= 128 in
23   f.i_accumulateur <- (2 * f.i_accumulateur) mod 256;
24   f.i_bits_accumules <- f.i_bits_accumules - 1;
25   bit
26 ;;
27
28 let close_in_bits f = close_in f.i_fichier ;;
```

Compression d'un octet

Pour pouvoir compresser un octet, il est nécessaire d'obtenir le chemin qui mène jusqu'à la feuille dont il est l'étiquette dans l'arbre de Huffman.

Pour cela, on commence par calculer l'ensemble des chemins de l'arbre de Huffman sous forme d'une table à 256 entrées qui contient le chemin associé à un octet s'il est présent dans l'arbre ou un chemin vide sinon.

On parlera de **représentation plate** de l'arbre de Huffman.

Il suffit de faire un parcours exhaustif de l'arbre pour réaliser cette table.

Compression d'un octet

```
let chemins a =
  let a_plat = Array.make 256 [] in
  let rec parcours a chemin =
    match a with
    | N (g, d) ->
      parcours g (false::chemin);
      parcours d (true::chemin)
    | F c -> a_plat.(c) <- List.rev chemin
  in parcours a [];
  a_plat
;;
```

```
let compresse_byte f a_plat c =
  let rec aux l =
    match l with
    | t::q -> output_bit f t; aux q
    | [] -> ()
  in aux a_plat.(c)
;;
```

Compression d'un octet

Afin de compresser un octet, on va donc aller lire le chemin dans cette table puis écrire le mot binaire correspondant grâce aux fonctions d'écriture bit à bit.

Compression d'un octet

Remarque

À chaque fois qu'on va compresser un octet, on va parcourir la liste correspondant à son chemin.

Comme les chemins les plus longs sont les moins fréquents, cela ne pose pas vraiment de problème.

Décompression d'un octet

```
1  let rec decompresse_byte f a = match a with
2    | F c -> c
3    | N(g,d) -> decompresse_byte f (if input_bit f then d else g)
4  ;;
```

Décompression d'un octet

Pour décompresser un octet, il suffit de parcourir l'arbre de Huffman en lisant bit à bit le fichier compressé, en descendant à gauche ou à droite selon si le bit lu est 0 ou 1.

Dès qu'on arrive sur une feuille, on écrit dans le nouveau fichier le caractère correspondant.

Compression de fichier

Algorithme final de compression

```
let compresse_fichier nom_in nom_out =
  let occ = table_occurrences nom_in in
  let a = construit_arbre occ in
  let f_out = open_out_bits nom_out in
  output_arbre f_out.o_fichier a;
  let a_plat = chemins a in
  let f_in = open_in_bin nom_in in
  begin
    try
      while true do
        let c = input_byte f_in in
        compresse_byte f_out a_plat c
      done
    with End_of_file -> ()
  end;
  close_in f_in;
  close_out_bits f_out
;;
```

Décompression de fichier

Algorithme final de décompression

```
let decompresse_fichier nom_in nom_out =
  let f_in = open_in_bits nom_in in
  let a = input_arbre f_in.i_fichier in
  let f_out = open_out_bin nom_out in
  begin
    try
      while true do
        let c = decompresse_byte f_in a in
        output_byte f_out c
      done
    with End_of_file -> ()
  end;
  close_in_bits f_in;
  close_out f_out
;;
```

Compression de fichier

Exemple

En compressant ainsi l'intégrale de Proust, on passe de 7543767 octets à 4249758 octets.

À titre de comparaison, l'outil unix `zip` permet d'obtenir un fichier de 2724213 octets.

Algorithme de Lempel-Ziv-Welch

LZW

L'algorithme d'Huffman est efficace, mais il présente un désavantage : il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal.

Dans cette partie, nous allons plutôt étudier une autre technique de compression qui, bien que moins efficace en pratique que Huffman, se programme assez facilement et permet de compresser des **flux** plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.

Il s'agit de l'algorithme de Lempel-Ziv-Welch, appelé communément **compression LZW**, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

Algorithme de Lempel-Ziv-Welch

Principe de la compression

L'idée de l'algorithme **LZW** est de faire avancer une fenêtre sur le texte en maintenant une table des motifs déjà rencontrés.

Quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.

Algorithme de Lempel-Ziv-Welch

Principe de la compression

Pour la table, on peut utiliser un **tableau dynamique** de motifs dont la taille ne pourra pas dépasser 2^d éléments, ou directement un **tableau** de 2^d **valeurs optionnelles**, dans la mesure où d est en général petit.

Ainsi, on pourra référencer chaque motif avec un mot de d bits.

Afin de retrouver efficacement l'indice du motif associé, on utilise une **table de hachage** réalisant l'inverse de la table.

Algorithme de Lempel-Ziv-Welch

Principe de la compression

L'algorithme procède alors ainsi pour compresser :

- on initialise la table avec une entrée pour chaque caractère, (donc chaque octet en considérant des caractères 8 bits) ;
- on maintient une variable contenant le plus long suffixe m du texte lu qui soit présent dans la table (il est initialisé avec la première lettre du texte) ;

Algorithme de Lempel-Ziv-Welch

Principe de la compression

L'algorithme procède alors ainsi pour compresser :

- on lit alors chaque caractère x :
  - soit mx est dans la table, et alors on remplace le motif courant par $m \leftarrow mx$;
  - soit mx n'est pas dans la table, (mais m y est d'après l'étape précédente) : on produit alors le code de m, on rajoute une entrée dans la table pour mx si elle contient moins de 2^d éléments, et on repart de $m \leftarrow x$;
- quand tous les caractères ont été lus, on produit le code correspondant à m.

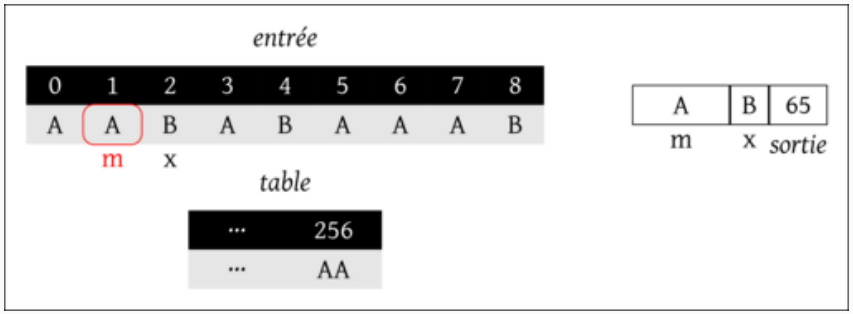
Algorithme de Lempel-Ziv-Welch

Exemple

Voici les différentes étapes pour la compression de "AABABAAAB" qui produit la suite d'entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, 'A' correspond à l'index 65, et 'B' à l'index 66.

Algorithme de Lempel-Ziv-Welch

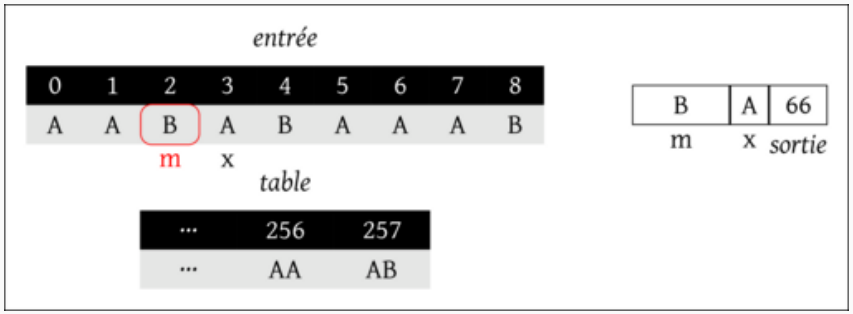


Exemple

Voici les différentes étapes pour la compression de "AABABAAAB" qui produit la suite d'entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, 'A' correspond à l'index 65, et 'B' à l'index 66.

Algorithme de Lempel-Ziv-Welch

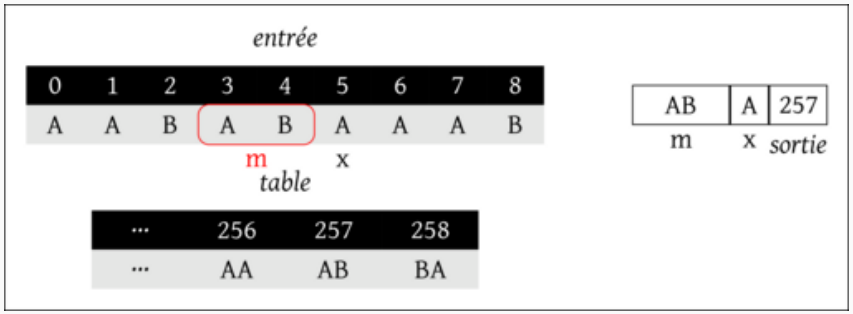


Exemple

Voici les différentes étapes pour la compression de “AABABAAAB” qui produit la suite d'entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, ‘A’ correspond à l'index 65, et ‘B’ à l'index 66.

Algorithme de Lempel-Ziv-Welch

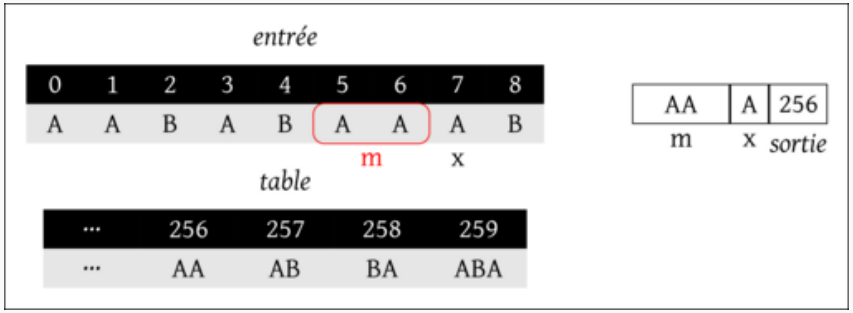


Exemple

Voici les différentes étapes pour la compression de “AABABAAAB” qui produit la suite d’entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, ‘A’ correspond à l’index 65, et ‘B’ à l’index 66.

Algorithme de Lempel-Ziv-Welch

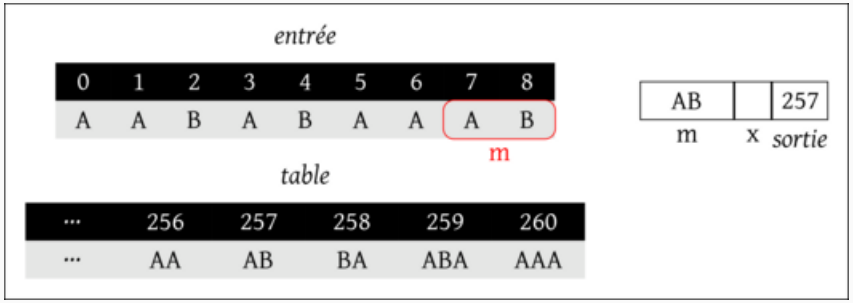


Exemple

Voici les différentes étapes pour la compression de "AABABAAAB" qui produit la suite d'entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, 'A' correspond à l'index 65, et 'B' à l'index 66.

Algorithme de Lempel-Ziv-Welch



Exemple

Voici les différentes étapes pour la compression de “AABABAAAB” qui produit la suite d’entiers $65, 256, 66, 65, 258, 257$, qui seront alors codés dans un fichier sur d bits.

Les 256 premières entrées correspondant au codage ASCII ; en particulier, ‘A’ correspond à l’index 65, et ‘B’ à l’index 66.

Algorithme de Lempel-Ziv-Welch

Principe de la décompression

Pour décompresser, on effectue la procédure précédente en sens inverse. Cependant, il faut reconstruire la table en même temps qu'on lit le fichier compressé.

Dans la majorité des cas, on va lire un code n tel que $n < |table|$, ce qui sera simple à gérer.

Cependant, il se peut qu'on lise parfois un code n tel que $n = |table|$. On verra comment gérer ce cas dans un second temps.

Algorithme de Lempel-Ziv-Welch

Principe de la décompression

Si tout se passe bien :

- pour le premier code lu, il s'agit forcément d'une référence à l'un des 256 caractères : on écrit donc ce caractère dans le fichier de sortie ;
- à partir du second code lu :
  - on lit un code n où $n < |table|$ et $table[n] = xm'$ avec x un caractère et m' un mot ;
  - on écrit m' dans le fichier de sortie ;
  - on rajoute ensuite mx dans la table, où $m = table[c]$ et où c est le précédent code lu.

Algorithme de Lempel-Ziv-Welch

Principe de la décompression

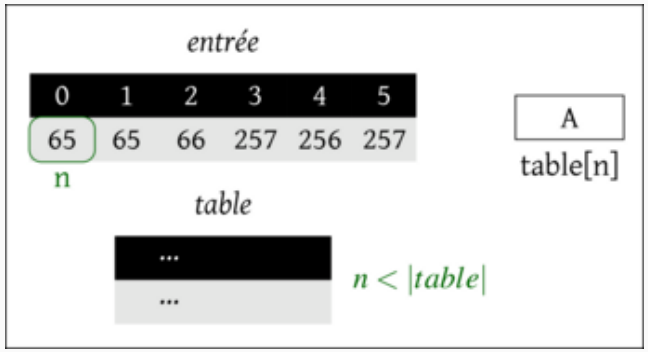
En faisant ainsi, on reproduit le processus de compression, mais en remplissant la table avec un temps de retard.

En effet, si on reprend le principe exposé plus haut, une entrée pour mx est ajoutée dans la table quand on lit le caractère x et que le motif lu précédemment est m, on repart alors avec x pour motif lu.

C'est exactement ce qu'on fait ici, en tenant compte du premier caractère de $table[n]$.

Contrairement à la compression, il est inutile ici de retrouver l'indice associé à un motif : on n'a donc pas besoin de la table de hachage utilisée lors de la compression.

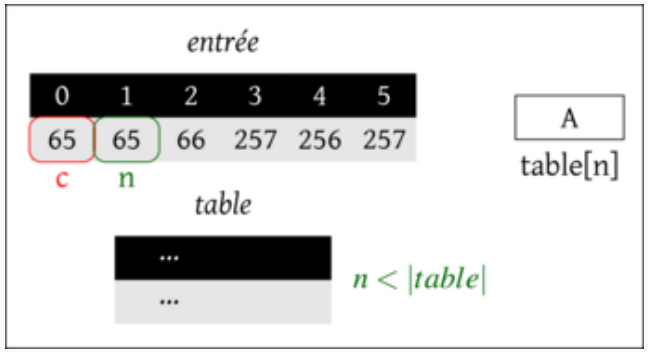
Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

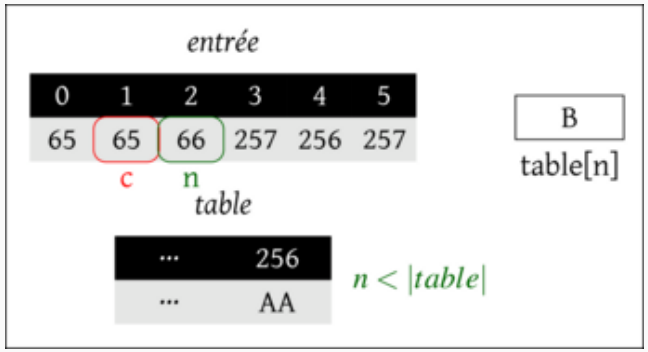
Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

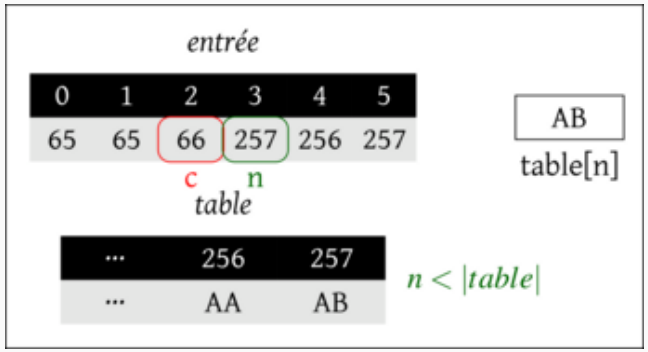
Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

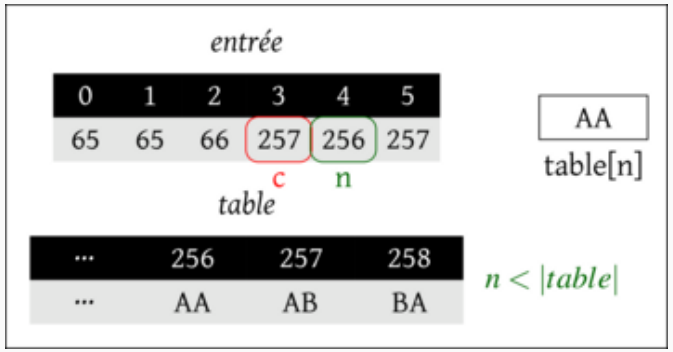
Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

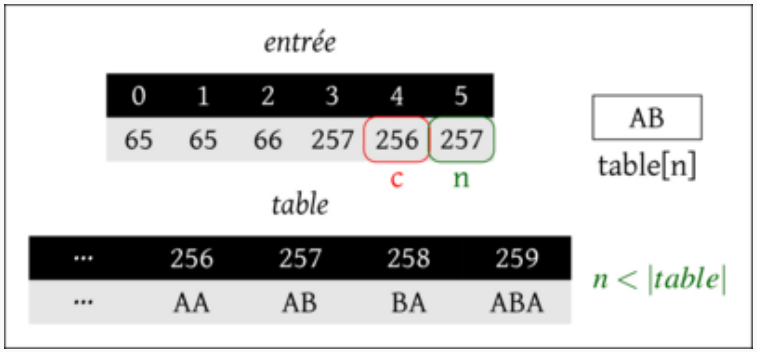
Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

Algorithme de Lempel-Ziv-Welch



Exemple

Voici les étapes de décompression sur l'exemple précédent.

Algorithme de Lempel-Ziv-Welch

Principe de décompression

Il reste toutefois un cas à traiter, celui où $n = |table|$, i.e. quand on lit un code qui n'est pas encore présent dans la table.

Pour comprendre ce cas, il est important d'identifier précisément quand il se produit dans le processus de décompression.

Algorithme de Lempel-Ziv-Welch

Principe de décompression

Comme on vient de le voir, on rajoute une entrée pour mx après avoir produit le code c correspondant à m.

Pour que le code n ne soit pas présent dans la table, il faut donc que n corresponde à cette entrée mx.

Or, quand on a compressé, on est reparti du motif x à ce moment là, donc m commence nécessairement par x.

Cela signifie qu'on peut reconstruire $table[n]$ en décompressant avec mx où x est la première lettre de $m = table[c]$.

Principe de décompression

On en déduit alors la procédure complète de décompression suivante :

- on initialise la table avec une entrée pour chaque caractère (comme pour la compression) ;
- on maintient une variable c contenant le dernier code lu, qu'on initialise avec le premier code en écrivant le caractère correspondant dans le fichier source ;
- pour chaque code n lu :
  - soit $n < |table|$ et $table[n] = xm'$ où x est un caractère, alors on écrit xm' en sortie ;
  - soit $n = |table|$ et alors on écrit en sortie $table[c]x$, où x est le premier caractère de $table[c]$;
  - dans tous les cas, on ajoute $table[c]x$ à la table, et on remplace c par n.

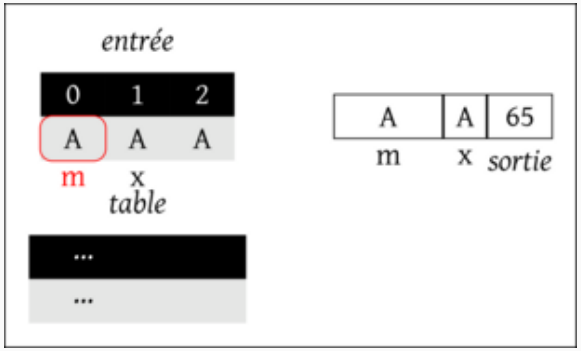
Algorithme de Lempel-Ziv-Welch

Exemple

Il y a un exemple classique où l'on a besoin de traiter ce cas : celui où le même caractère est présent plusieurs fois en début de fichier.

Comme **LZW** est utilisé pour compresser des formats d'images où les pixels sont des indices dans une palette de 256 couleurs avec une couleur pour la transparence, le cas où la même couleur est présente au début du fichier est fréquent.

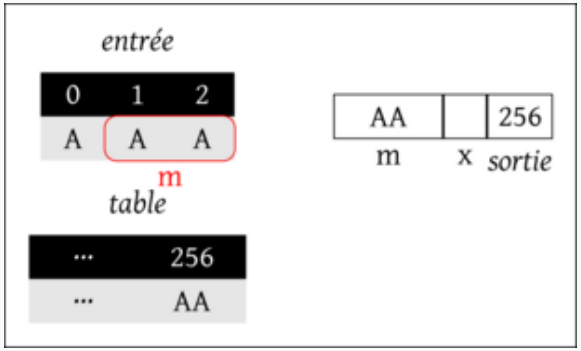
Algorithme de Lempel-Ziv-Welch



Exemple

Voici un exemple de compression et de décompression pour le texte “AAA”.

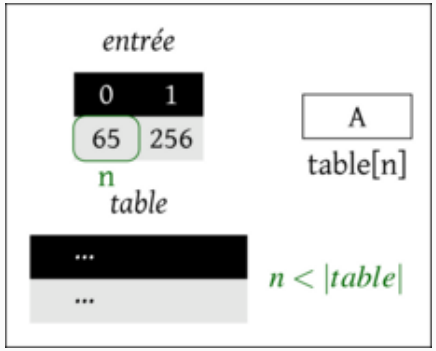
Algorithme de Lempel-Ziv-Welch



Exemple

Voici un exemple de compression et de décompression pour le texte “AAA”.

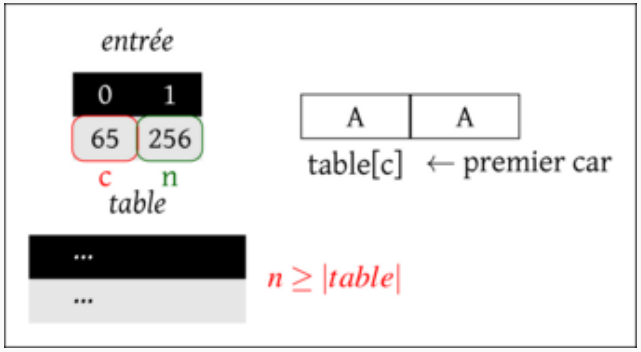
Algorithme de Lempel-Ziv-Welch



Exemple

Voici un exemple de compression et de décompression pour le texte “AAA”.

Algorithme de Lempel-Ziv-Welch



Exemple

Voici un exemple de compression et de décompression pour le texte “AAA”.

Algorithme de Lempel-Ziv-Welch

```
1  let input_code f d =
2    let acc = ref 0 in
3    for i = 0 to d - 1 do
4      let b = input_bit f in
5      if b
6      then acc := !acc + (1 lsl i)
7    done;
8    !acc
9  ;;
10
11 let output_code f code d =
12   assert (code < 1 lsl d);
13   let acc = ref code in
14   for i = 0 to d - 1 do
15     output_bit f (!acc mod 2 = 1);
16     acc := !acc / 2
17   done
18 ;;
```

Implémentation en OCaml

Tout d'abord, il est nécessaire de fournir des fonctions de manipulation des entiers sur d bits, et de lecture/écriture dans un fichier.

Algorithme de Lempel-Ziv-Welch

Compression (1/2)

```
type table_bidir = {
  elements : string array;
  indices : (string, int) Hashtbl.t;
  mutable n_elements : int
} ;;

let string_of_byte b = String.make 1 (Char.chr b) ;;

let cree_table d =
  let taille_table = 1 lsl d in
  let elements = Array.make taille_table "" in
  let indices = Hashtbl.create taille_table in
  for i = 0 to 255 do
    let s = string_of_byte i in
    elements.(i) <- s;
    Hashtbl.add indices s i
  done;
  { elements = elements; indices =  indices; n_elements = 256 }
;;

let ajoute_entree table s =
  table.elements.(table.n_elements) <- s;
  Hashtbl.add table.indices s table.n_elements;
  table.n_elements <- table.n_elements + 1
;;
```

Algorithme de Lempel-Ziv-Welch

Compression (2/2)

```
let compresse_fichier nom_in nom_out d =
  let table = cree_table d in
  let taille_table = 1 lsl d in
  let f_out = open_out_bits nom_out in
  let f_in = open_in_bin nom_in in
  let m = ref (string_of_byte (input_byte f_in)) in
  begin
    try
      while true do
        let c = string_of_byte (input_byte f_in) in
        let mc = !m ^ c in
        match Hashtbl.find_opt table.indices mc with
        | Some _ -> m := mc
        | None -> begin
            let i = Hashtbl.find table.indices !m in
            output_code f_out i d;
            if table.n_elements < taille_table
            then ajoute_entree table mc;
            m := c
          end
      done
    with End_of_file -> ()
  end;
  let i = Hashtbl.find table.indices !m in (* dernier code *)
  output_code f_out i d;
  close_in f_in;
  close_out_bits f_out
;;
```

Algorithme de Lempel-Ziv-Welch

Décompression

```
let decompresse_fichier nom_in nom_out d =
  let table = cree_table d in
  let taille_table = 1 lsl d in
  let f_in = open_in_bits nom_in in
  let f_out = open_out_bin nom_out in
  let code = ref (input_code f_in d) in
  output_string f_out table.elements.(!code);
  begin
    try
      while true do
        let nouveau = input_code f_in d in
        let s_code = table.elements.(!code) in
        let s =
          if nouveau = table.n_elements
          then let x = String.sub s_code 0 1
            in s_code ^ x
          else table.elements.(nouveau) in
        let x = String.sub s 0 1 in
        output_string f_out s;
        if table.n_elements < taille_table
        then ajoute_entree table (s_code ^ x);
        code := nouveau
      done
    with End_of_file -> ()
  end;
  close_in_bits f_in;
  close_out f_out
;;
```

Algorithme de Lempel-Ziv-Welch

Exemple : Impact de la longueur du code

Afin d'étudier l'impact de la longueur du code sur la taille des fichiers compressés, on considère deux fichiers :

- `proust.txt` contenant, en 7543768 octets, l'intégrale de *à la recherche du temps perdu* de Marcel Proust.
- `code.py` contenant 8566 octets de code source Python.

On note **np** la taille en nombre d'octets après compression du fichier `proust.txt` et **tp** le nombre d'entrées dans la table à la fin du processus.

De même, on note **nc** et **tc** les valeurs respectives pour le fichier `code.py`.

Algorithme de Lempel-Ziv-Welch

d	np	nc	tp	tc
8	7543768	8566	256	256
9	5056398	5709	512	512
10	4195124	4431	1024	1024
11	3864174	3948	2048	2048
12	3612505	4039	4096	2947
13	3434424	4376	8192	2947
14	3262145	4712	16384	2947
15	3131790	5049	32768	2947
16	2998639	5385	65536	2947
17	2682264	5722	131072	2947
18	2554323	6058	262144	2947
19	2500314	6395	524288	2947
20	2557656	6731	1023317	2947

Exemple : Impact de la longueur du code

On obtient alors les valeurs ci-dessus (avec différents d).

Algorithme de Lempel-Ziv-Welch

Exemple : Impact de la longueur du code

On constate qu'il est nécessaire d'avoir un texte riche pour bénéficier d'une grande longueur de code.

Le fichier `code.py` ne contenant pas plus que 2947 motifs.

Même si le fichier `proust.txt` en contient plus que les tailles considérées ici, il y a un compromis qui s'établit entre la richesse de la table et la taille du code.

Ainsi, il semble que le fichier `proust.txt` soit compressé de manière optimale avec $d = 19$.