Projet codage de texte

Formation ISN Professeurs de Terminale

Denis Bouhineau, Éric Gaussier, Alexandre Termier, Cyril Labbé, Philippe Bizard, Anne Rasse, Jean-Marc Vincent

UFR IM^{2}AG
email Jean-Marc.Vincent@imag.fr

Formation professeurs de terminale Niveau I

Plan

Objectifs du projet

1. Analyse d'algorithme et conception
   → arbres, files à priorité, itérations récursion
2. Écriture de code en langage C
   → renforcer la pratique de ce langage, fichiers, co-développement
3. Validation à chaque étape
   → analyse de complexité, évaluation de performances (expérimentation)

- Structure de mini-projet :
  - travail en équipe : binômes
  - temps plein (sur une semaine)
  - autonomie
- Évaluation : soutenance/démonstration du logiciel
  - Déroulement :
    - démonstration : $\sim$ 15 mn
    - questions : $\sim$ 15 mn
  - Objectifs : montrer ce qui fonctionne
    - préparer des jeux d'essai
    - prévoir le déroulement de la présentation
    - travailler le discours

Organisation

Planning :

organisation: TD de 8h30 à 9h30, salle machine ensuite

lundi : **démarrage**	[salle TD]
mardi : **architecture de projet**	[salle machine]
mercredi : **entrées/sorties bit à bit**	[salle TD]
jeudi : **développement**	[salle machine]
vendredi : **validation, analyse de coût, démo**	[salle TD]

Encadrement :

Denis Bouhineau, Éric Gaussier, Alexandre Termier, Cyril Labbé, Philippe Bizard, Anne Rasse, Jean-Marc Vincent

page Web :

Thème du projet : compression sans pertes

Au menu :

- Algorithme de Huffman
  - programes de codage et décodage
  - lecture/écriture de fichiers binaires
  - compression/décompression “à la volée”
- Algorithme *Run Length Encoding*
  - version de base
  - variantes (historique, séquence d'échappement, PackBits)
- Expérimentation sur divers formats de données :
  textes, programmes (source et binaire), images, etc.

Rappels : codage de Huffman

- alphabet d'entrée $\mathcal{A}$ (caractères, pixels, etc.)
  alphabet de sortie $\{0,1\}$ (bit)

- un code de Huffman :
  associe à chaque élément de $\mathcal{A}$ une séquence sur $\{0,1\}$:

$$\mathcal{A} \overset{codage}{\longrightarrow} \{0,1\}^* \qquad \{0,1\}^* \overset{decodage}{\longrightarrow} \mathcal{A}$$

- caractéristiques :
  - code de longueur variable (= code Morse ; $\neq$ code Ascii)
  - propriété de préfixe :
    codage(a_1)=w_1, codage(a_2)=w_2
    $\Rightarrow$ w_1 et w_2 ne sont pas préfixes l'un de l'autre
    ($\rightarrow$ algorithme de décodage efficace)

Application du codage de Huffman

Coder une séquence d'éléments sur $\mathcal{A}$ dans un fichier binaire :

- associer un code "court" aux élements les "plus fréquents"
- $\rightarrow$ **compression** de la séquence initiale

Remarques :

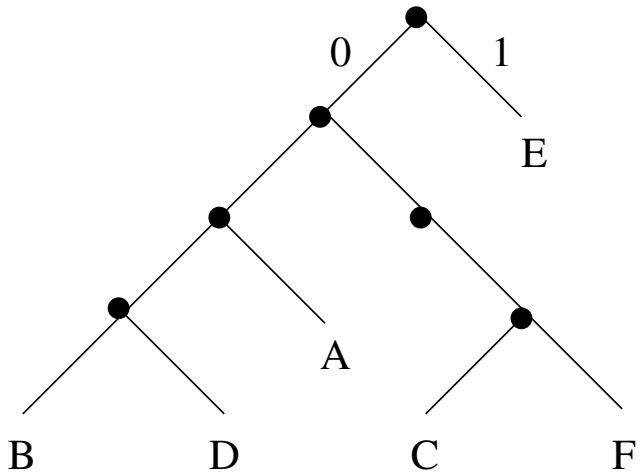
- le code doit être construit **en fonction** du texte initial ...
  - **avant** le codage : Huffman "statique"
  - **pendant** le codage : Huffman "dynamique"
- il doit être connu lors du décodage ...
  (transmis ou recalculé à l'identique)

Arbre de Huffman

Une représentation d'un code de Huffman = arbre de Huffman

- ensemble des feuilles = éléments de $\mathcal{A}$
- accès au fils gauche = "0" ; accès au fils droit = "1"

code(a) = séquence σ des étiquettes du chemin : racine $\xrightarrow{\sigma} a$

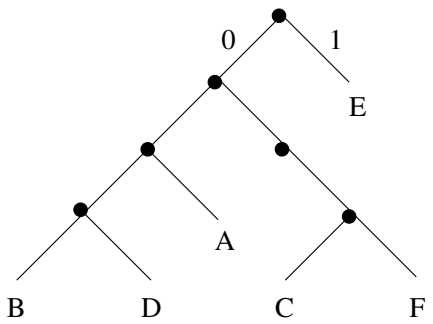


code(E)=1 ; code(A)=001 ; code(C)=0110 ; etc.
$\rightarrow$ la propriété de préfixe est respectée !

Algorithme de Codage

$$S = a_1 \dots a_n \rightarrow R \in \{0,1\}^*$$

Etant donné un arbre de Huffman :



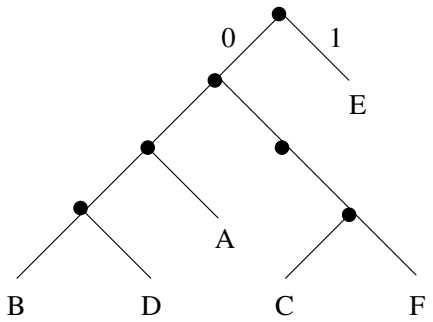
1. construction d'une table de codage `C`
   - parcours en "profondeur d'abord" de l'arbre de Huffman en mémorisant la séquence σ "racine $\xrightarrow{\sigma}$ noeud courant"
   - pour chaque feuille a : `C[a]` $= \sigma$

   Rq : algo récursif ou itératif (avec pile explicite)
2. parcours de S et production de R à l'aide de `C`

Algorithme de Décodage

$$R \in \{0,1\}^* \rightarrow S = a_1 \dots a_n$$

Etant donné un arbre de Huffman :



1. parcours de l'arbre de Huffman selon R
   ("0" : fils gauche ; "1" : fils droit)
2. à chaque feuille a atteinte :
   - écriture de a dans S
   - retour à la racine de l'arbre de Huffman ...

Construction d'un arbre de Huffman ?

- Dépend de la fréquence des éléments de $\mathcal{A}$ dans S :
  fréquence élevée =¿ éléments proches de la racine

- Obtention d'une table de fréquences : `Freq`$[a_i] = f_i$
  (f_i = fréquence de a_i dans S, un réel entre 0 et 1)

- Approche "statique" :
  1. construction de `Freq` par un 1er parcours de S
     (f_i = nombre d'occurences de a_i / $\mid S \mid$)
  2. construction de l'arbre de Huffman (à l'aide de `Freq`)

Algorithme informel de construction de l'arbre

On utilise une **file à priorités** (Fap) `F` dont :

- les éléments sont des noeuds de l'arbre
- les priorités sont des réels (valeur élevée $\Rightarrow$ priorité forte)

Algo :

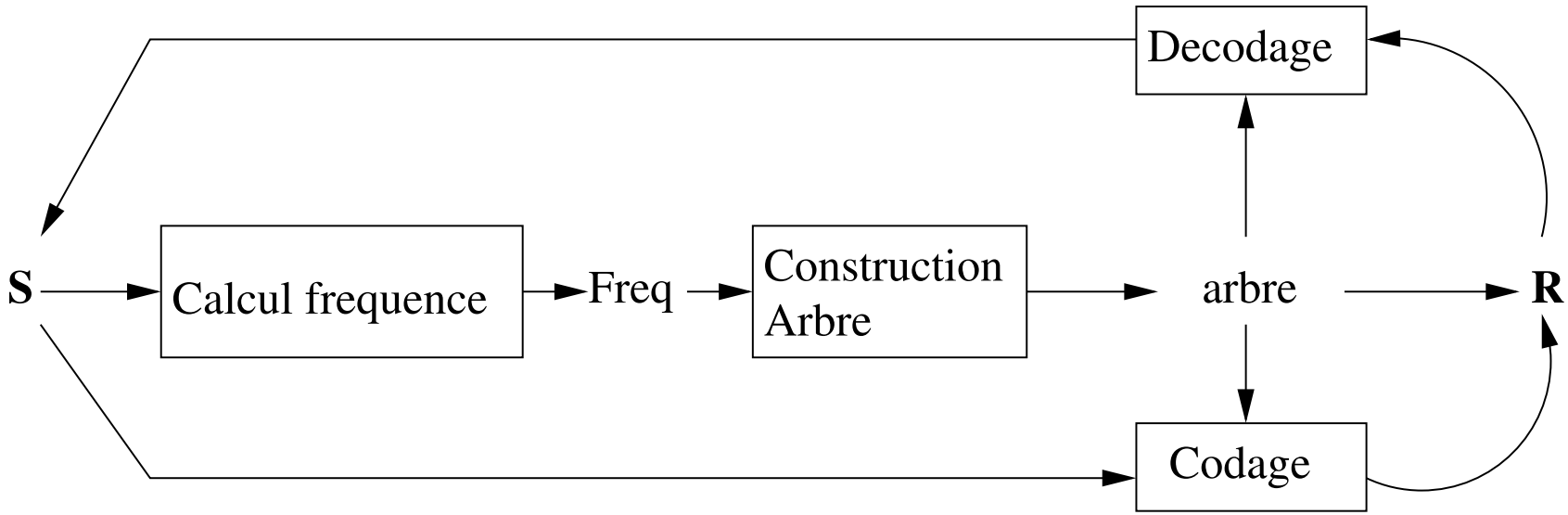
1. insérer les éléments (a_i, `Freq[`a_i`]`) dans la fap `F`
2. tantque `F` contient au moins 2 éléments
   - extraire deux éléments (n_1, f_1) et (n_2, f_2) de poids minimal de `F`
   - insérer $(n, f_1 + f_2)$ dans `F`, où n est un nouveau noeud de fils gauche n_1 et de fils droit n_2
3. l'élément restant dans `F` est la racine de l'arbre de Huffman

Structures de données

- Ensemble $\mathcal{A}$: caractères d'un code Ascii étendu (0 à 255)
- Table `Freq` : tableau de réels indicé de 0 à 255
- Arbre de Huffman :
  - soit chaînage de cellules par pointeurs
  - soit tableau (256 feuilles $\Rightarrow$ 511 noeuds)
- Fap `F` : séquence de couples (noeud de l'arbre, réel)
- Séquence sur $\{0,1\}^*$: suite de caractères '0' et '1'

Architecture globale

Un ensemble de programmes interconnectés :



→ communications possible par fichiers, ou “tubes” (*pipe*) . . .

Exemple de découpage en commandes

- Commande de calcul des fréquences :
  `calcul_frequences < entree.txt > frequences.txt`
- Codage (construit l'arbre puis code) :
  `codage frequences.txt < entree.txt > code.txt`
- Décodage (construit l'arbre puis décode) :
  `decodage frequences.txt < code.txt > sortie.txt`
- Vérification du résultat :
  `diff entree.txt sortie.txt`
- Taille du texte d'entrée :
  `stat -f '%z' entree.txt`
- Taille du texte codé :
  `stat -f '%z' code.txt`

Notions complémentaires : fichiers

- Manipulés par leur nom dans l'interpréteur de commandes
- Accès en C via un descripteur : structure contenant les informations sur l'accès en cours
- Utilisation :
  - Ouverture → obtention d'un nouveau descripteur
    ```
    FILE *fopen(char *nom, char *mode);
    ```
  - Accès / Test
    ```
    int fscanf(FILE *f, char *format, ...);
    int fprintf(FILE *f, char *format, ...);
    int feof(FILE *f);
    ```
  - Fermeture → libération des ressources
    ```
    int fclose(FILE *f);
    ```

Exemple : lecture et affichage d'un fichier

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    FILE *fichier; char c;

    if (argc > 1) {
        fichier = fopen(argv[1], "r");
        if (fichier != NULL) {
            fscanf(fichier, "%c", &c);
            while (!feof(fichier)) {
                printf("%c", c);
                fscanf(fichier, "%c", &c);
            }
        }
    }
    return 0;
}
```

Infos utiles sur les fichiers

- Mode : "r" ou "w", mais d'autres existent
- En cas d'erreur d'ouverture, utiliser `perror`

```
void perror(char *message);
```

- Descripteurs ouverts par défaut :
  - *stdin* : entrée standard (clavier)
  - *stdout* : sortie standard (écran)
  - *stderr* : sortie d'erreur standard (écran)

Exemple : écriture dans un fichier

```
#include <stdio.h>
int main() {
    FILE *fichier; char c;

    fichier = fopen("toto.txt", "w");
    if (fichier != NULL) {
        scanf("%c", &c);
        while (!feof(stdin)) {
            fprintf(fichier, "%c", c);
            scanf("%c", &c);
        }
    } else {
        perror("Erreur à l'ouverture");
    }
    return 0;
}
```

Extension 1 : lecture/écriture binaire

R doit être en binaire (valeurs 0 et 1 codées sur un bit)

Problème : l'élément minimal géré avec les fichiers est l'octet
Solution : mise en tampon

- grouper 8 bits dans un tampon de 1 octet (type `char`)
- `void ecrire (FILE *f, Tampon T, char b)` :
  - ajouter le bit `b` à `T` (ex: en décalant `T` à gauche et en plaçant `b` en bit de poid faible de `T`)
  - si `T` est plein, écrire l'octet correspondant dans f
- `char lire (FILE *f, Tampon T)` :
  - si `T` est vide, lire `T` depuis le fichier
  - lire le prochain bit de `T` (ex: bit de poids fort puis décaler `T` à gauche pour l'accès suivant)

Attention : Les bits doivent être relus dans le même ordre que lors de l'écriture (comme dans l'exemple donné)

Lecture/écriture binaire (suite)

- Idéalement, il faut encapsuler la gestion du tampon :
  - définir une structure `BINARY_FILE`
  - fonction d'ouverture pour obtenir un `BINARY_FILE` à partir d'un descripteur de fichier
  - fonctions d'accès, test et fermeture

- Problème pour les séquences de bits non multiples de 8 :
  - définir une séquence d'échappement (ex: 0xFF)
  - Fin de la séquence : 0xFF suivi du nombre de bits restant et du tampon incomplet
  - Lors de l'écriture de 0xFF, le coder par 0xFFFF et en tenir compte lors de la lecture

Extension 2: compression “à la volée”

Inconvénient de la sol. “statique” :
le codage nécessite 2 parcours de S
Une approche “dynamique” : construire `Freq` **lors** du codage !

- initialiser `Freq` avec une valeur identique pour chaque élement de $\mathcal{A}$ ($f_i = 1/ \mid \mathcal{A} \mid$) et construire un arbre initial
- pour chaque nouvel élément e de S :
  - coder e à l'aide de l'arbre courant
  - mettre à jour `Freq`
  - construire un **nouvel** arbre à partir de `Freq`

Rq : l'algo de décodage suit le même principe
(reconstruction de l'arbre après chaque caractère décodé)

Contenus adaptés à la compression RLE

Tout contenu comprenant une suite d'octets identiques :

- Sources en divers langages : indentation (espaces + tab)
- Plages uniformes d'images en format sans compression
  maxi 1 octet/pixel : niveaux de gris/couleurs 8 bits
- .o et exécutables : section data (valeurs initiales nulles) et champs à 0 dans tables.
- Pages de mémoire partiellement vides : des 0 à la fin.

Run Length Encoding : principe

Une suite de 15 octets consécutifs identiques dans une séquence de 16 caractères :

S = "abbbbbbbbbbbbbbb" ('a' et 15 fois 'b')

peut être réduite à 3 octets : S = 'a','b',#, 14

- Le 1^{er} octet des octets identiques : 'b'
- un marqueur spécial : # indiquant une répétition
- un nombre de répétitions : 14 (octet : max$<$ 255)
- séquence trop longue = suite de séquences courtes.

Choix du marqueur de répétition

Cas simple : fichier de texte (ASCII, iso-latin1, utf8, ...).
$\rightarrow$ utiliser pseudo-caractères (contrôle affichage ou télécoms).

Exemple : codes Ctrl-Q/Ctrl-S de contrôle de flux Xon/Xoff
codes ASCII 0x17 (DC1/Xon) et 0x19 (DC3/Xoff)

Cas général : contenu binaire quelconque, toutes les valeurs d'octets peuvent faire partie du contenu.

Méthode simple : [x,x,n] représente [x, ..., x] (n+2 fois x)
$\rightarrow$ deux octets consécutifs identiques = marqueur spécial.

Exemples

s1="abbbbcdddeffg0h" et s2="abcdeeegga"

0 : caractère '0', $\underline{0}$: entier 0 (nb répétitions)

a b b b b c d d d e f f g 0 h

↓ rle

a b b $\underline{2}$ c d d $\underline{1}$ e f f $\underline{0}$ g 0 h

a b c d e e e g g a

↓ rle

a b c d e e $\underline{1}$ g g $\underline{0}$ a

Taux de compression selon longueur

n codé sur b⩽8 bits (1 octet): $nmax = 2^b - 1$.
3 octets RLE xxn encodent séquence x...x (L=n+2 fois).

- L=1 : (noir) codage RLE sur 1 octet (0%)
- L=2 : (rouge) codage RLE sur 3 octets (perte 1, 50%)
- L=3 : (vert) codage RLE sur 3 octets (0%)
- L=4 : (jaune) codage RLE sur 3 octets (gain 1, 25%)
- L=5 : (jaune) codage RLE sur 3 octets (gain 2, 40%)

bits de n (b)	2	3	4	5	6	7	8
nmax ($2^b - 1$)	3	7	15	31	63	127	255
Lmax=nmax+2	5	9	17	33	65	129	257
Gain (%) $\frac{Lmax-3}{Lmax}$:	40	67	82	91	95	98	99

Optimiser les séquences de longueur 2

Inconvénient de marqueur = 2 valeurs identiques :

nb_séquences (s,n=2) > nb_séquences (s,n>2)

peut donner

longueur(rle(s)) > longueur (s)

Objectif : optimiser le codage des séquences xxy ($x \neq y$).

- coder répétitions sur 7 bits suffit (gain maxi 98%)
- n > 0 : code de x...x (>2 fois)
- n < 0 : code de xxy avec y dans E avec Card(E) = 2^7.
- n = 0 : code de xx

Plus de perte sur xxy pour 50% des valeurs possibles de y

Historique de taille limitée

Choisir E tel que $y \in E$ pour un maximum de xxy :

1. Sans information : choix statique, programmation simple
   Ex : pari sur ASCII (E = [0,127]).
2. Avec information déjà accumulée :
   pari sur des valeurs déjà vues → historique

Principe d'un historique de taille limitée :

- Card(E) valeurs, un exemplaire (le + récent) de chaque
- de la séquence ou de rle(séquence), numérotées
- de la plus récente (0) à la plus ancienne (Card(E)-1)
- inclure ou non nb répétitions : histo(s) ou histo(rle(s)).

Exemples : historique de s1, rle(s1), s2, rle(s2)

Lors du codage de xxy, on cherche y dans l'historique de la partie de séquence déjà traitée (terminée par xx). Au final :

8				7	6			5	4		3	2	1	0
a	b	b	b	b	c	d	d	d	e	f	f	g	0	h

11		10	9	8		7	6	5		4	3	2	1	0
a	b	b	$\underline{2}$	c	d	d	$\underline{1}$	e	f	f	$\underline{0}$	g	0	h

	5	4	3			2		1	0
a	b	c	d	e	e	e	g	g	a

	7	6	5		4	3		2	1	0
a	b	c	d	e	e	$\underline{1}$	g	g	$\underline{0}$	a

Application de RLE avec historique

Exemple d'utilisation de l'historique sur la séquence RLE.

Comparaison de rle sans (s) et avec (a) historique.

(s)	a	b	b	2	c	d	d	1	e	f	f	0	g	0	h
(a)	a	b	b	2	c	d	d	1	e	f	f	0	g	0	h

n = 0 : la lettre g n'appartient pas à l'historique

(s)	a	b	c	d	e	e	1	g	g	0	a
(a)	a	b	c	d	e	e	1	g	g	-6	
	-6	-5	-4	-3		-2	-1		0		

n<0 : la lettre g est dans l'historique au rang |n|=6

A faire impérativement !

- version de “base” du codage de Huffman
  - version “statique”
  - S et R dans des fichiers Ascii
- évaluation sur divers formats de données
  (HTML, source de programme, binaire, images, etc.)
- comparaison des **taux de compression** obtenus

Diverses extensions à la carte . . .

- lecture/écriture binaire
- compression “à la volée”
- RLE sans et/ou avec historique
- combinaisons Huffman / RLE
- compression à partir de mots
- compression d’images
- évaluation de ces diverses extensions

Choix du sujet...

- Richesse et importance du problème
  → historique, codage, compression...
- Difficultés graduées
  → réussite et challenge (moteurs)
- Appel à des compétences différentes
  → programmation, analyse,
- Travail en autonomie
  → support adapté