Polytech Grenoble - RICM3

Projet en langage C

Du 22 au 30 mai 2017

Encadrement : Philippe.Waille@imag.fr

Objectifs du projet

- Ecriture de code en langage C
  $\rightarrow$ renforcer la pratique de ce langage

- Structure de mini-projet :
  - travail en équipe : quadrinômes
  - temps plein (sur une semaine)
  - autonomie

- Evaluation : soutenance/démonstration du logiciel

Organisation

Planning :

- Présentation : lundi 22/05 matin : A101
- Salles : 013,**035,039,043**,257
- Soutenances mardi 30 mai après-midi : salles en gras

Resources :

- Documents et code fournis : Alfresco Share (pointé dans moodle)
- Machine de référence pour le code fourni : mandelbrot

Soutenances/Démonstration

- Déroulement :
  - démonstration : $\sim$ 15 mn
  - questions : $\sim$ 10 mn
- Objectifs : montrer ce qui fonctionne (thème : démo au client)
  - préparer des jeux d'essai
  - prévoir le déroulement de la présentation
  - travailler le discours
- Que présenter :
  - les fonctionnalités et l'utilisation $\neq$ présenter code
  - doc : comment compiler/utiliser, limites et bugs
  - structuration en modules, éventuellement extrait de code spécifiquement intéressant
- Rendre : une archive de votre code (tar ou zip par mail).

Thème : compression sans pertes

- Algorithme LZW (Lampel-Ziv-Welch)
  - programes de codage et décodage
  - réalisation d'un dictionnaire
  - lecture/écriture de fichiers binaires
  - compression/décompression "à la volée" (| compresser > f.z)

- Expérimentation sur divers formats de données :
  textes, programmes (source et binaire), images, etc.

- Attention : contenu binaire : valeurs 0 et 255 légales.
  - Pb si dictionnaire = tableau de chaînes C (octet de contenu 0 pris comme fin de chaîne)
  - EOF est souvent -1 $\rightarrow$ 255 si tronqué à 8 bits :
    while ((c=getchar()) != EOF) $\Rightarrow$ c de type int et non char

Notion de compression sans perte

- Une méthode de représentation de données dans un code C1.
- Une méthode de représentation des mêmes données dans un autre code C2.
- La transformation C1 vers C2 est intégralement réversible : le passage de C1 à C2 ne perd aucune information sur les données.
- Le codage en C2 occupe moins de mémoire que le codage en C1.

Exemples contraires en audio et vidéo (mp3, jpeg, etc) : compression avec pertes des détails "non perceptibles".

Dictionnaire

- Données à compresser = éléments d'un dictionnaire, un élément $\rightarrow$ un code.
- Séquence des éléments $\rightarrow$ séquence des codes.
- **Huffman** : longueur des codes des éléments selon fréquence dans le fichier à compresser. Elément typique : octet.
- Approche **Lampel Ziv** : dictionnaire construit dynamiquement selon contenu du fichier. Elément = séquence d'octets
- $\forall$ méthode : contenu compressé = flux de bits et non d'octets. $\rightarrow$ problème de détection de la fin du flux compressé (dernier octet partiellement significatif).

Approche Lampel Ziv Welch

Principe : encoder des séquences de un ou plusieurs octets.

- Dictionnaire $\notin$ fichier compressé : **construit dynamiquement** de la même manière par émetteur et récepteur
- $\forall$ seq séquence **mono-octet**, seq $\in$ dictionnaire initial.
- Les **séquences** multi-octets dont le **préfixe** appartient déjà au dictionnaire y sont **ajoutées dynamiquement**.
- Les éléments du dictionnaire sont numérotés et tous les **numéros** codés sur le **même nombre de bits**.
- Nombre de bits des codes **incrémenté dynamiquement** quand la taille du dictionnaire atteint une **nouvelle puissance** de 2.

Lampel Ziv Welch (1984)

Welch améliore les algorithmes de Lampel et Ziv : LZ77 et LZ78.
LZW : breveté, brevet tombé depuis dans le domaine public (2004).

Décomposition de toute séquence en seq_préfixe + seq_mono.
Exemple : "lion" = "lio" (préfixe) + "n" (mono)

Compression : chaque séquence est décomposée en un doublet (p,f).
p : numéro de préfixe (mono- ou multi- octet)
f : numéro de suffixe (un seul octet)

Recherche et codage de la plus grande séquence pf dont le préfixe p soit déjà présent dans le dictionnaire, puis ajout de la séquence pf dans le dictionnaire.

Technique d'apprentissage efficace sur répétition de longues chaînes.

Exemple : chaînes débutant par " l" (espace+'l')

258	32+'l'	Le_lion et_le rat
259	258+'e'	
260	259+' '	Il faut, autant qu'on peut, obliger tout_le_monde :
		On a souvent besoin d'un plus petit que soi.
		De cette vérité deux fables feront foi,
261	258+'a'	Tant_la chose en preuves abonde.
262	259+'s'	Entre_les pattes d'un_lion
263	258+'i'	
264	258+"'"	Un rat sortit de terre assez à_l'étourdie.
		Le Roi des animaux, en cette occasion,

258	259	260	261	262	263	264
_l	_le	_le	_la	_les	_li	_l'

265	258+'u'	Montra ce qu'il était, et lui donna la vie.
		Ce bienfait ne fut pas perdu.
		Quelqu'un aurait-il jamais cru
266	263+'o'	Qu'un lion d'un rat eût affaire ?
		Cependant il advint qu'au sortir des forêts
267	266+'n'	Ce lion fut pris dans des rets,
268	260+'p'	Dont ses rugissements ne le purent défaire.
		Sire rat accourut, et fit tant par ses dents
269	264+'o'	Qu'une maille rongée emporta tout l'ouvrage.
		Patience et longueur de temps . . .

258	259	260	261	262	263
l	le	le	la	les	li

264	265	266	267	268	269
l'	lu	lio	lion	le p	l'o

Exemple : chaînes débutant par "on"

Le lion et le rat

Il faut, autant qu'on peut, obliger tout le monde :
On a souvent besoin d'un plus petit que soi.
De cette vérité deux Fables feront foi,
Tant la chose en preuves abonde.
Entre les pattes d'un lion
Un rat sortit de terre assez à l'étourdie.
Le roi des animaux, en cette occasion,

"on" "on " "ond" "ont" "onde" "on \n" "on,"

"on" "on " "ond" "ont" "onde" "on \n" "on,"

Montra ce qu'il était, et lui donna la vie.
Ce bienfait ne fut pas perdu.
Quelqu'un aurait-il jamais cru
Qu'un lion d'un rat eût affaire ?
Cependant il advint qu'au sortir des forêts
Ce lion fut pris dans des rets,
Dont ses rugissements ne le purent défaire.
Sire rat accourut, et fit tant par ses dents
Qu'une maille rongée emporta tout l'ouvrage.
Patience et longueur de temps
Font plus que force ni que rage.

"ontr" "onn" "on d" "on f" "ont " "ong" "ongu" "ont p"

Gains et pertes du codage LZW

- LZW utilise un dictionnaire de $2^{(8+p)}$ éléments.
- Une séquence normale de L octets occupe 8L bits.
- Une séquence lzw de $L > 1$ octets **absente** du dictionnaire ($\simeq$ L séquences mono) occupe L fois (8+p) bits : **expansion**.
- Une séquence lzw de L octets **trouvée** dans le dictionnaire occupe (8+p) bits : **compression**.

Long	Dic	$\Delta \rightarrow$ %	p=1	p=2	p=3	p=4
$L > 1$	$\notin$	$\frac{p+8}{8}$	+12,5	+25	+37,5	+50
L=1						
L=2		$\frac{p+8}{8L}$	-43,7	-37,5	-31,2	-25
L=3	$\in$		-62,5	-58,3	-54,1	-50
L=4			-71,8	-68,7	-54,1	-66,6

Algorithmes et Structures de Données

Algorithme de codage

lexique :

E : fichier d'entrée ; S : fichier de sortie

D : un dictionnaire (ens. de chaînes d'octets identifiées par un index)

w : chaîne d'octets ; a : un octet

algo :

$D \leftarrow$ ens. de toutes les chaînes de longueur 1 ; $w \leftarrow$[1er octet de E]

tant que la fin de E n'est pas atteinte

$a \leftarrow$ octet suivant de E

si $w.a$ est dans D **alors**

$w \leftarrow w.a$ { recherche d'une plus longue sous-chaîne }

sinon

{w est la plus longue sous-chaîne présente dans D}

écrire sur S l'index associé à w dans D

$D \leftarrow D \cup \{w.a\}$

$w \leftarrow [a]$

écrire sur S l'index associé à w

Algorithme de décodage (version préliminaire)

lexique :

S : fichier d'entrée ; E : fichier de sortie

D : un dictionnaire (ens. de chaînes d'octets identifiées par un index)

i, i' : index dans D

w, w' : chaînes d'octets ; a : un octet

algo :

$D \leftarrow$ ens. de toutes les chaînes de longueur 1

$i \leftarrow$ 1er code de S ; $a \leftarrow$ chaîne d'index i dans D ; $w \leftarrow [a]$

écrire w sur E

tant que la fin de S n'est pas atteinte

$i' \leftarrow$ code suivant de S ; $w' \leftarrow$ chaîne d'index i' dans D

écrire w' sur E

$a \leftarrow$ 1er octet de w'

$D \leftarrow D \cup \{w.a\}$

$i \leftarrow i'$; $w \leftarrow$ chaîne d'index i dans D

Un problème potentiel (1)

Séquence d'entrée : `xxxza ...`

Codage :

lecture de `x` ; lecture de `x` ; chaîne courante = `x`

→ écriture du code de `x` (120) ; ajout de `xx` (259) dans D

lecture de `x` ; lecture de `z` ; chaîne courante = `xx`

→ écriture du code de `xx` (259) ; ajout de `xxz` (260) dans D

lecture de `a` ; chaîne courante = `z`

→ écriture du code de `z` (122) ; ajout de `za` (261) dans D

Contenu du fichier codé : 120.259.122 ... (x.xx.z. ...)

Un problème potentiel (2)

Fichier codé : `120.259.122 ...`

Décodage :

lecture de 120 ; caractère courant = `x`

→ écriture de `x` (120)

lecture de 259

→ **code non présent dans** D **!**

⇒ il s'agit nécess. du car. courant concaténé au dernier code lu : `xx`

→ écriture de `xx` ; ajout de ajout de `xx` (259) dans D

lecture de 122 ; caractère courant = `z`

→ écriture de `z` ; ajout de `xxz` (260) dans D

Fichier décodé : `xxxz ...`

Algorithme de décodage (version corrigée)

$D \leftarrow$ ens. de toutes les chaînes de longueur 1
$i \leftarrow$ 1er code de S ; $a \leftarrow$ chaîne d'index i dans D ; $w \leftarrow [a]$
écrire w sur E
tantque la fin de S n'est pas atteinte
    $i' \leftarrow$ code suivant de S
    si $i' \notin D$ **alors**
        $w' \leftarrow$ chaîne d'index i dans D
        $w' \leftarrow w'.a$
    sinon
        $w' \leftarrow$ chaîne d'index i' dans D
    écrire w' sur E
    $a \leftarrow$ 1er octet de w'
    $D \leftarrow D \cup \{w.a\}$
    $i \leftarrow i'$; $w \leftarrow$ chaîne d'index i dans D

Exemple : compression de "abcabcabcde"

- $\underrightarrow{ab}cabcabcde$: 96(a) émis, dict += "ab"/258, départ = 'b'
- $ab\underrightarrow{ca}bcabcde$: 97(b) émis, dict += "bc"/259, départ = 'c'
- $abca\underrightarrow{bc}abcde$: 98(c) émis, dict += "ca"/260, départ = 'a'
- $abcabca\underrightarrow{bc}abcde$: 258(ab) émis, dict += "abc"/261, départ = 'c'
- $abcabcab\underrightarrow{cd}e$: 260(ca) émis, dict += "cab"/262, départ = 'b'
- $abcabcabc\underrightarrow{de}$: 259(bc) émis, dict += "bcd"/263, départ = 'd'
- $abcabcabcd\underrightarrow{eEOF}$: 99(d) émis, dict += "de"/264, départ = 'e'
- $abcabcabcde\underrightarrow{EOF}$: 100(e) émis, 256(FIN) émis

Compressé : 96,97,98,258,260,259,99,100,256

Exemple : décompression de "abcabcabcde"

- 96 : "**a**" émis
- 97 : "**b**" émis, dict += "ab"/258
- 98 : "**c**" émis, dict += "bc"/259
- 258 : "**ab**" émis, dict += "ca"/260
- 260 : "**ca**" émis, dict += "abc"/261
- 259 : "**bc**" émis, dict += "cab"/262
- 99 : "**d**" émis, dict += "bcd"/263
- 100 : "**e**" émis, dict += "de"/264
- 256 : FIN . . . de la décompression

Exemple de format de sortie pour "abcdabd"

Sans e final et avec marque de fin

Séquence des numéros codés sur 9 bits et groupement en octets

97	("a")	0x061	0 0110 0001	0011 0000	0x30
98	("b")	0x062	0 0110 0010	1001 1000	0x98
99	("c")	0x063	0 0110 0011	1000 1100	0x8c
100	("d")	0x064	0 0110 0100	0110 0110	0x66
259	("ab")	0x103	1 0000 0011	0100 1000	0x48
100	("d")	0x064	0 0110 0100	0001 1001	0x19
256	(END)	0x100	1 0000 0000	1001 0010	0x92
			0	0000 000**0**	0x00
					1 bit padding

"abcdabd END " → contenu de fichier = 30 98 8c 66 48-19-92-00

Le dictionnaire

→ Mémorise des couples (Séquence d'octets, Code)

Exemple d'interface possible :

- `void Initialiser()`
  initialise un dictionnaire avec toutes les monoséquences
- `Code Inserer (Code prefixe, Code mono)`
  ajoute la séquence d'octets `prefixe.mono`, renvoie son `code`
- `uint8_t *CodeVersChaine (Code code, int *longueur)`
  renvoie séquence et affecte la longueur associée à `code`
- `Code SéquenceVersCode (uint8_t *séquence, int longueur)`
  renvoie le code associé à `séquence`

Propriétés :

- une séquence n'est insérée qu'après tous ses préfixes
- pas de suppression individuelle...

Exemples de structures de données possibles

1. tableau de séquences d'octets
   - recherche séquentielle ; insertion en queue
   - peu efficace, mais **facile à programmer !**
   - version triée : recherche dichotomique
2. arbre binaire de recherche (de séquences d'octets)
   - améliore le coût de la recherche
3. table de hachage
   - hachage fermé ou ouvert (listes de collisions)
   - choix d'une "bonne" fonction de hachage ?
4. dictionnaire arborescent
   - partage de nombreux préfixes communs
   - recherche et insertion "efficaces"

Le travail demandé

A faire impérativement !

- version de “base” du codage LZW
  - version “simple” du dictionnaire
  - compression+décompression au minimum simulée en format ASCII.
- Fichier résultat = suite de bits des codes de séquences
  Résultat simulé = suites de (numéros,taille_en_bits) écrits en ASCII.
- évaluation sur divers formats de données
  (HTML, source de programme, binaire, images, etc.)
- estimation des **taux de compression** : même en simulation ASCII on peut compter le nombre de bits de la suite de codes.

Diverses extensions à la carte . . .

- lecture/écriture binaire des codes (vraie compression)
- versions plus efficaces (temps, espace mémoire) du dictionnaire (arbres ternaires, table de hachage, etc.)
- adaptation dynamique de la taille des codes (nombre de bits) en fonction de la taille du dictionnaire (++ à chaque nouveau 2^x).
- ré-initialisation partielle ou totale du dictionnaire (si plein, si taux de compression stagne, . . .)
- Run Length Encoding ?
- évaluation de ces diverses extensions

Conseils

Les ressources fournies

- LZW : un binaire fonctionnel et instrumenté (traces d'exécutions)
  → pour comprendre l'algorithme !
- DICT : une implémentation de dictionnaire arborescent
  → pour démarrer et comme base de comparaison
- BINIO : un module d'entrées/sorties
  → pour démarrer : lire ou écrire en binaire des entiers sur x bits
- une assistance en salles de TP et/ou par mail
  → pour répondre à vos questions . . .

Organisation du travail (suggestions)

- prendre le temps de **comprendre** le problème, les ressources fournies avant de se lancer dans le code et le découpage en modules.
- Quadrinômes + structure du projet
  $\Rightarrow$ **répartition** du travail (ex : équipes de 2)
- Bien définir les **interfaces** entre les différents modules (en-tête de fonction, formats de fichiers)
- Compiler et **tester** séparément chaque module !
- Intégrer et sauvegarder régulièrement des versions "stables" (mêmes incomplètes) : git et tar sont vos amis !
- Garder du temps pour :
  - faire des mesures expérimentales
  - préparer la soutenance !

Détails techniques : gestion de taille des codes

Contenu des dictionnaires de émetteur et récepteur du flux compressé :

- évolution théoriquement synchronisée
- 2^{x+1} séquences atteintes : nombre_bits_des_codes ++ $\rightarrow$ x+1
- nombre_de_bits **parfaitement** synchronisé, même sur code inconnu au décodage ?

- Prévoir éventuellement réinitialisation partielle ou totale du dictionnaire (trop gros $\rightarrow$ chute du taux de compression).

Détails : détection de fin de contenu compressé

Le dernier octet du contenu compressé peut contenir des bits de remplissage. Parmi les méthodes d'arrêt possibles :

- préfixer le flux compressé par la taille du contenu initial : simple mais peu compatible avec lecture directe du contenu à compresser sur entrée standard
  Ex : ./ma_cde | ./compresseur > trace_sortie_ma_cde.lzw
- après lecture du dernier octet, calculer le nombre de bits non consommés du flux compressé. Si $\geq$ **longueur_courante** des codes de séquences, lire les **longueur_courante** bits du dernier code et émettre la séquence de contenu décompressé.
- prévoir explicitement un symbole de fin de contenu

Une méthode simple et générale

Pour ces problèmes, suggestion d'une méthode générique simple :

- Ajouter aux 256 valeurs possibles d'octet du dictionnaire initial des séquences fictives de gestion de l'algorithme de compression.

- Fin de contenu
- Incrémentation de la taille des codes
- Réinitialisation du dictionnaire

- Dictionnaire initial de $(256 + 3)$ monoséquences $\rightarrow$ taille initiale des codes $= 9$ bits.

Mesure de compression

Préciser l'indicateur de mesure :

1) rapport de taille : compressé = X

$Taille\_fichier\_compressé / Taille\_fichier\_initial$

2) taux de compression : supression de X

$1 - Taille\_fichier\_compressé / Taille\_fichier\_initial$