Projet de Théorie de l'information M1 2011-2012.
Compression de données.

1 Description du sujet

La compression de données permet d'économiser de la place sur les unités de stockage. Le but de ce projet est d'implanter la méthode de Lempel et Ziv (LZ78). Il s'agit ici d'écrire une commande `lempelziv` qui compresse ou décompresse un fichier dont le nom lui est passé en argument selon qu'elle est appelée avec l'option -c ou -d.

1.1 La méthode de Lempel et Ziv

La méthode consiste à coder une suite de caractères de la manière suivante : cette suite est découpée en phrases. Chaque phrase est constituée d'une phrase rencontrée précédemment (la plus longue possible) plus un caractère. Les phrases sont numérotées (dans l'ordre où elles sont rencontrées) et chacune d'elle est codée par le numéro de la phrase précédente sur laquelle elle est construite plus le caractère à rajouter. La phrase vide a pour numéro 0.

Voici un exemple :

texte initial :	aaabbabaabaaabab						
phrases :	a	aa	b	ba	baa	baaa	bab
numéro de phrase :	1	2	3	4	5	6	7
texte codé :	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Attention, les caractères ne sont pas forcément des char ni forcéments des bits. On pourra paramétrer le programme de sorte à considérer des caractères codés sur 1 bit (alphabet à deux lettres), 2 bits (alphabet à 4 lettres), 3 bits (alphabet à 8 lettres), ... , 8 bits (alphabet à 256 lettres). D'autre part, le nombre k de phrases déjà rencontrées est variable ; aussi pour économiser sur la taille du codage des numéros de phrase, la phrase courante est codée par un caractère et le numéro d'une des k phrases déjà rencontrées est codé sur $\log_2 k$ bits.

1.2 Décompression

Un des avantages de cette méthode est la rapidité de la décompression. En effet il suffit de stocker dans un tableau toutes les phrases déja rencontrées pour pouvoir décoder immédiatement la phrase courante.

Une optimisation utile pour limiter la taille de cette table consiste à ne stocker pour chaque phrase que son code (ce qui suffit pour retrouver la phrase caractère par caractère).

(Cette partie doit impérativement être traitée après la suivante.)

1.3 Compression

Pour que la compression soit rapide, il faut utiliser une structure de données arborescente appelée trie. Le problème consiste en effet à trouver rapidement

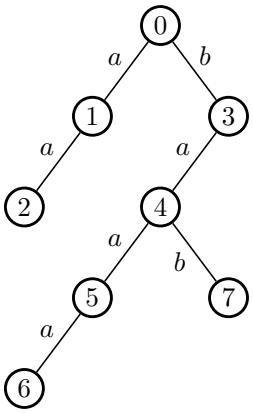

FIG. 1 – Le trie obtenu après lecture du texte "*aaabbabaabaaabab*".

le numéro de la plus longue phrase rencontrée précédemment qui se retrouve à la position courante de lecture dans le fichier. Un trie répond à cette attente, et il permet de stocker toutes les phrases rencontrées précédemment sous forme d'arbre.

Chaque phrase correspond à un noeud de l'arbre. La phrase 0 qui représente la suite vide est la racine de l'arbre. Les fils de chaque noeud sont numérotés par une lettre (on dit que cette lettre est l'étiquette de l'arc entre le père et le fils). Un noeud de l'arbre correspond à la phrase obtenue en lisant les étiquettes sur les arcs du chemin de la racine au noeud. Le numéro de la phrase est stocké dans le noeud correspondant de l'arbre. (Un noeud peut avoir autant de fils qu'il y a de lettres dans l'alphabet.)

Le moyen le plus simple de comprendre cela est d'examiner le trie obtenu après la lecture du texte de l'exemple vu plus haut (voir la figure 1).

Le traitement de la phrase courante est le suivant. On lit les lettres du texte à partir de la position courante une à une et on parcourt le trie en partant de la racine en même temps. Chaque fois que le fils du nœud ou l'on se trouve correspondant la lettre lue existe, on se positionne à ce nœud. Si le fils n'existe pas, on le crée et on s'arrête là, c'est le noeud qui correspond à la phrase courante.On a donc lu une phrase de plus et on peut passer à la suivante. Par exemple, si le texte de la figure 1 se poursuit par "*babbbaab*", on rajoutera la phrase 8 "*babb*" comme fils d'étiquette *b* du noeud 7 et la phrase 9 "*baab*" comme fils d'étiquette *b* du nœud 5. Pour les gros fichiers, il ne faut pas saturer la mémoire avec un trop gros trie. Une stratégie consiste à recommencer à zéro quand on a lu un certain nombre de caractères. Une autre consiste à recommencer à zéro quand le trie atteint une certaine taille. Cette éventuelle optimisation doit être prise en compte lors de la décompression. L'idéal est de maintenir l'espace mémoire utilisé par le programme en dessous d'un certain seuil, 1 Mo par exemple.

2 Calendrier

Le projet est à réaliser en Java.

Le projet est à faire en binôme (cela veut dire deux personnes et pas trois ni une, sauf si vous êtes un nombre impair). Le projet est à rendre avant le 4 mars 2012 par email dont le titre est "Projet ThInfo M1" sous la forme d'une pièce jointe au format zip contenant l'ensemble des programmes et des documents. Le email devra être envoyé aux deux adresses `beal@univ-mlv.fr` et `lombardy@univ-mlv.fr`. Le fichier doit s'appeller `nom1_nom2.zip`, où nom1 et nom2 sont les noms des binômes dans l'ordre alphabétique.

Les optimisations ne sont pas obligatoires. Si vous manquez de temps, le principal est d'avoir un programme qui tourne avec un alphabet à 2 lettres.

3 Détail du rendu

Voici les noms des répertoires et fichiers qui doivent être contenus dans l'archive zippée.

- Un ficher `readme.txt` indiquant comment recompiler et exécuter le programme, ou se trouve la documentation, etc.
- Un script shell `lempelziv` permettant de lancer le programme avec les options voulues.
- Un répertoire `src` contenant les sources `.java`.
- Un répertoire `bin` contenant les classes `.class`.
- Un répertoire `doc` contenant
  - La documentation utilisateur sous forme d'un document pdf `user.pdf` contenant, en plus des informations classiques (comment compiler, exécuter, etc ...) une description de l'application et comment l'utiliser.
  - La documentation développeur `dev.pdf` contenant une description détaillée de chaque classe que vous avez implémentée. Il faut essayer de décrire ici les structures de données et les algorithmes. On indiquera ici la liste des bugs connus, s'il y en a, avec un scénario permettant de générer le bug ainsi que la raison pour laquelle le bug se produit. Ce document pourra contenir des tests d'exécution. Vous pourrez notamment vous intéresser au taux de compression (taille du fichier compressé/taille du fichier). Des tests comparatifs avec différentes tailles d'alphabet et différentes limitations de mémoire seront grandement appréciés.
  - Un sous-répertoire `api` contenant la documentation complète au format javadoc.

4 Appendice : maniement des données bit à bit

Pour la lecture et l'écriture de bits en Java, on pourra utiliser les classes `BitInputStream` et `BitOutputStream` accessibles à l'adresse suivante : `http://igm.univ-mlv.fr/~beal/Java/Programmes/Bits/` .