Examen Codage Transmission Compression M2 Image 2008-2009

Le 15/12/2008, Tout document autorisé

Recopier tel quel des extraits de documents n'apporte jamais de points. Vous devez répondre aux questions qui vous sont posées. Vous devez argumenter (étayer) vos réponses. Une réponse correcte non étayée ne rapporte pas de points, et une réponse incorrecte bien étayée peut en rapporter.

1. Dans quels cas faut-il utiliser LZ77 ?

2. Quels liens voyez-vous entre les interfaces utilisateur et la compression ?

3. Quel est le taux maximum de compression possible quand on veut faire de la compression temps réel ?

4. Voici un codage possible de l'unicode :

0xxxxxxx	Un code $< 2^7$
100xxxxx xxxxxxxx	Un code $< 2^{13}$
101xxxxx xxxxxxxx xxxxxxxx	Un code $< 2^{21}$
110xxxxx xxxxxxxx xxxxxxxx xxxxxxxx	Un code $< 2^{29}$
111xxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx	Un code $< 2^{37}$

   Donnez les avantages et inconvénients de ce codage par rapport à l'UTF8.

5. Vous manipulez une hiérarchie de petits fichiers (moins de 10 octets). On suppose qu'il n'y a pas de liens *hard*. Comment le système d'exploitation devrait stocker cette hiérarchie de fichiers pour avoir :
   (a) une place occupée minimale.
   (b) un temps d'accès minimal en lecture.
   (c) une place et un temps presque minimal pour la lecture.
   (d) une place et un temps presque minimal pour l'écriture.

6. On veut faire de la compression en temps constant, quels sont les méthodes qui posent un problème ?

7. Pour numériser fidèlement une peinture (un tableau artistique) on prend des photos avec différents points de vues et différentes sources de lumière. Comment comprimer l'ensemble de ces photos ?

8. Dans quels cas la méthode de Huffman fait-elle un équilibrage parfait des probabilités ?

9. Faut-il ajouter un code de vérification d'erreur sur les données avant ou après la compression ? Donnez **toutes** les raisons.

10. Soit x le coût énergétique pour avoir un méga-octet de donnée accessible pour un an. Soit y le coût énergétique pour décomprimer ou décomprimer un méga-octet. Quelle stratégie adopter pour minimiser le coût énergétique ?

11. Comment comprimeriez-vous des images avec 5 canaux : RVB+IR+UV ?

12. Proposez une méthode de compression adaptée aux sources de programmes écrits en langage C.
    (a) Avec ou sans pertes.
    (b) Générant un code compilable ou non.
    (c) Permettant ou non d'améliorer le programme.

Une correction

1. On utilise LZ77 quand :

   - La suite d'octet à comprimer contient des répétitions. (elle n'est donc pas bruitée)
   - On a besoin d'une décompression utilisant peu de temps CPU.
   - La suite d'octet à comprimer est très redondante **localement** ceci permet d'avoir un tout petit buffer et de coder les positions sur peu de bits.
   - La suite d'octet contient des répétitions longues comme ***********...* ou *RVBRVBRVBRVBRVB...* que LZ77 va comprimer en faisant du RLE.

2. L'interface utilisateur véhicule des informations entre les applications et l'utilisateur. Pour maximiser le débit de ce canal de communication il faut comprimer les données qui le traverse.

   - Pour l'affichage, il faut minimiser la redondance des informations qui sont sur l'écran et il faut trouver des représentations des informations qui minimisent la place sur l'écran tout en étant naturellement compréhensible.
   - Pour la gestion des événements, il faut calculer leur probabilité de déclenchement et associer aux événements les plus probables l'action la plus rapide à faire pour l'utilisateur.

     Il faut aussi déterminer les événements dépendants pour qu'ils soient faciles à déclencher l'un après l'autre. Par exemple rapprocher les boutons de déplacement bas/haut de l'ascenseur car ils sont souvent utilisés l'un après l'autre.

3. La compression temps réel implique que l'on envoit l'événement comprimé dès que celui-ci est disponible (on attend pas le suivant) et que si le réseau est utilisé on ne tient pas compte du passé (à cause des paquets perdus).

   Le taux maximum de compression temps réel est le nombre de bits de l'événement que l'on désire comprimer. Dans ce cas, on envoit un bit pour transmettre l'événement. On ne peut pas faire mieux car cela voudrait dire que l'on serai obligé d'attendre un deuxième événement avant d'envoyer le code compressé.

4. Ce codage a l'avantage de perdre moins de place que l'unicode, mais

   - Il ne permet pas l'accès aléatoire.
   - Il n'est pas compatible avec le langage C et les systèmes de fichiers car les caractères NUL et / peuvent apparaître n'importe où.
   - On ne peut pas détecter automatique ce format car toute suite aléatoire d'octets représente un codage valide. A l'exception des derniers octets.

5. Hiérarchie de petits fichiers.

   - Place minimale : on stocke la hiérarchie comme une suite d'événements respectant la grammaire simplifiée :

     ```
     ListeNoeud : {NULL} | (Noeud ListeNoeud)
     Noeud : NomFichier ListeAttribut
     NomFichier : une chaine de caractère
     ListeAttribut : {NULL} | (Attribut ListeAttribut)
     Attribut : Fils | Contenu | UID | GID | Mode | DateModification
     ```

```
UID : {UID} un entier
GID : {GID} un entier
Mode : {MODE} un entier
DateModification : {DATEMODIFICATION} un entier
Contenu : {CONTENU} une chaine de caractères
Fils : {FILS} ListeNoeud
```

Pour les UID/GID/Mode on n'indique que les changements par rapport au fichier précédent, la plupart du temps ces attributs ne changent pas.

Un fichier vide, n'a pas d'attribut 'contenu'.

Une entité est un répertoire car elle a un attributs 'fils'.

Les identificateurs {...} sont codés dans leur propre flux, donc il y aura plus de bits pour indiquer les attributs rarement utilisés.

A chaque type attribut est associé son propre flux qui compressera intelligemment la donnée en question. Par exemple les dates sont codées par rapport à la date précédente. Les noms de fichier dans un dictionnaire de nom de fichier...

Il est certainement intéressant d'avoir les tags MODE A+X et MODE A-X qui seraient sans paramètre, ce sont les changements de mode les plus courants.

- L'accès minimal en lecture interdit l'usage de l'accès séquentiel. L'utilisation d'une table de *hashage* minimise le temps d'accès au fichier car il est constant en fonction du nombre de fichiers. La table de *hashage* fournit un pointeur sur une structure de taille variable définissant le fichier :

  ```
  NomCourt PointeurPère ListeAttribut
  ```

  Les attributs sont définit par différence par rapport au père au lieu du fichier précédent. Il n'est plus possible d'avoir un flux pour la compression des valeurs, elles sont stockées indépendemment les unes des autres en utilisant un dictionnaire si possible.

  L'attributs fils des répertoires contiendra des pointeurs sur les fils.

- Place et temps presque minimal pour la lecture. On reprend le premier schéma mais en utilisant des pointeurs.

  ```
  ListeNoeud : {NULL} | (Noeud ListeNoeud)
  Noeud : NomFichier Pointeur
  ```

  Les attributs sont relatifs au père. Si les répertoires contiennent de très nombreux fichiers, il ne faut pas utiliser une liste mais un arbre.

- Place et temps presque minimal pour la l'écriture. On est ici obligé d'utiliser des pointeurs pour ne pas avoir à décaler les données en mémoire. Les deux méthodes précédentes ont des temps d'accès en écriture équivalents. Dans les deux cas on peut être obligé de recopier la donnée ailleurs.

  Le pire cas est la modification sur un attribut du père qui implique la modification des attributs de tous les fils. Il est donc certainement intéressant de stocker les attributs sans faire référence à une valeur précédente bien que cela prenne de la place.

Une autre solution pour les 3 derniers points est d'utiliser 2 structures de données complètement différentes. Une au niveau supérieur qui optimise la vitesse d'accès (arbre) et une optimale en place au niveau inférieur qui permet par exemple de stocker un sous arbre complet sur un seul bloc disque.

6. Compression en temps constant :
   - Pas de problème pour l'équilibrage de probabilité, les temps de calcul sont fixes et bornés par la longueur du codage de l'événement.
   - LZ77 : S'il est programmé simplement, certaines chaines de caractères peuvent faire exploser le temps de recherche.
   - LZ78, LZW8 : Il faut oublier des chaînes sinon le temps augmente.
   - JPEG, Ondelette : Pas de problème.
   - La partie psycho acoustique de la compression de son peux poser un problème quand aucune fréquence n'écrase les autres, on es alors obligé de tout parcourir.
   - MPEG : Limiter le temps de recherche de bloc similaire.
   - Block sorting : Des fichiers particuliers peuvent faire exploser le temps du *Move to Front*.
7. Il faut reconstituer un modèle du tableau avec pour chaque "pixel" sa hauteur (épaisseur de peinture) et ses coefficients de réflexion de la lumière. Ce modèle sera comprimé comme une série d'image contenant chacune un coefficient. La notion HLS n'est d'aucune utilité ici car les coefficients sont indépendants.

   Chaque photo est alors définie par un point de vue, les différentes sources de lumière et éventuellement la différence avec l'image réelle si aucune perte n'est tolérée.
8. L'équilibrage est parfait quand les probabilités des événements sont des inverses de puissances de 2. En effet, dans ce cas, à chaque noeud de l'arbre binaire la coupure est parfaite.

   Évidemment avoir des événements équiprobables n'est pas suffisant. Quel arbre avez-vous avec 3 événements ?
9. On ajoute le code de détection d'erreur après la compression car :
   - Car un changement d'un bit dans le code comprimé a toutes les chances de déclencher la modification de plusieurs bits dans le code décomprimé, ceci rendant le code de détection d'erreur inefficace car il est lui même modifié.
   - Le code de détection n'est pas comprimable car il est "aléatoire".
   - On est forcé de décomprimer avant de pouvoir faire un acquitement pour indiquer que l'on a bien reçu la donnée.
   - Cela prend moins de temps pour calculer le code correcteur car le fichier est plus petit.

   Si l'on ne fait pas confiance à la méthode de compression on calcule le code sur le fichier original et non sur la donnée comprimée. Dans ce cas, on perd les deux derniers avantages à moins de calculer/stocker 2 codes...
10. Pour une donnée de un méga octet définie par :
    - Soit n le nombre d'accès moyen par année.
    - Soit c le taux de compression.

    Coût de stockage non compressé : x
    Coût de stockage compressé : $cx + 2yn$
    Il faut donc comprimer si : $n < \frac{x(1-c)}{2y}$
    On peut optimiser en gardant la version comprimée pour ne pas la recomprimer à nouveau si elle n'a pas changée, ceci fait disparaître le facteur 2.

11. On comprime évidemment en jpeg ou ondelette. Il faut traduire les couleurs en luminance plus une teinte incluant les ultra-violet (YUVW). Les infra rouges sont bien moins liés à la couleur, il est très probable qu'il est plus intéressant de leur dédier un canal à part, d'autant plus que ce canal contient beaucoup moins de hautes fréquences et l'algorithme de compression peut avoir des paramètres différents.

12. Le compresseur extraira l'arbre syntaxique du programme source et réécrira cet arbre avec un codage compacte qui sera ensuite comprimé avec un compresseur classique.

    Les noeuds de l'arbre syntaxique sont extrêmement contextuel, il y a donc peu de choix à chaque niveau. Un arbre d'équilibrage statique par contexte convient donc parfaitement.

    - Par défaut l'arbre syntaxique perd les commentaires et espacements. Si on veut les conserver, il faut les ajouter ce qui prend une place considérable.
    - On peut renommer les variables et fonctions internes au programmes pour *réutiliser les mêmes noms* et donc permettre d'avoir un meilleur taux de compression. Il est possible que cette action empèche la compilation du programme s'il utilise des macros astucieuses. Raccourcir les noms de variable ne fait pas gagner grand chose car le compresseur qui suit élimine cette redondance grâce à son dictionnaire.
    - On peut à partir de l'arbre syntaxique trouver dans l'arbre des parties identiques après canonisation et ainsi créer automatiquement des fonctions qui manquaient au programme. Ceci peut dans certain cas empêcher le fonctionnement du programme (introspection, multi thread...)

    Si un compresseur classique est utilisé tel quel il ne faut pas que le codage compacte décrivant l'arbre syntaxique descende au niveau du bit mais se limite au niveau de l'octet. Sinon le compresseur classique n'éliminera même pas la redondance au niveau des noms de variables.