LOURD Rodolphe
Génie Informatique

CODAGE DE HUFFMAN EN JAVA

Objectif

Le but de ce projet est de coder ou de décoder une suite de caractères à l'aide d'une séquence binaire et d'un arbre de Huffman avec des règles de construction définies. Ainsi, il sera possible de compresser un texte donné par l'utilisateur en une suite de chiffres binaires, ou l'inverse.

1. Présentation

Un codage de Huffman d'une chaîne de caractères est constitué d'un couple (arbre de décodage, séquence binaire codée). L'arbre de décodage est un arbre binaire dont les feuilles sont étiquetées par des caractères. Chaque feuille (caractère) est codée par une séquence binaire de la façon suivante : si l'on va vers le fils gauche, on ajoute 0 à la séquence ; si l'on va vers le fils droit, on ajoute 1 à la séquence. Ainsi chaque caractère présent dans l'arbre sera représenté de manière unique par un code binaire (séquence) qui ne pourra être préfixe d'un code d'un autre caractère (pour éviter les erreurs lors du décodage).

2. Manipulation d'arbres binaires

On va tout d'abord définir l'objet *arbre* avec ses méthodes nécessaires à sa manipulation/construction.

a. Constructeurs du type arbre

```
arbre(char c)
  {
  gauche=null;
  droite=null;
  lettre=c;
  }

arbre(arbre g,arbre d,char c)
  {
  gauche=g;
  droite=d;
  lettre=c;
  }

arbre(arbre v)
  {
  lettre=v.lettre;
  gauche=v.gauche;
  droite=v.droite;
  }
```

Avec de tels constructeurs, on peut créer une feuille *(arbre(char c))*, créer un noeud *(arbre(arbre g,arbre d,char c))* et faire une copie d'arbre *(arbre(arbre v))*.

b. Fonction estfeuille

```
public static boolean estfeuille(arbre n)
  {
  if((n.droite==null)&&(n.gauche==null)) return (true);
  else return (false);
  }
```

Cette méthode permet de savoir si l'on est en bout d'arbre.

c. Procédure imprime_arbre

Ici on a les différentes méthodes d'affichage d'un arbre ; essentiellement pour vérifier sa construction.

```
public static void imprime_arbre(arbre n,int d)
    {
    if (n==null) {}
    else
      {
      imprime_arbre(n.droite,compteur++);
      compteur--;
      String s="";
      for (int i=0;i<compteur;i++) s+=" ";
      n.afficher(s);
      imprime_arbre(n.gauche,compteur++);
      compteur--;
      }
    }
```

Cette méthode affiche un arbre selon la première représentation, c'est-à-dire en cascade avec une rotation de 90° horaire.

```
public static void imprime_texte_arbre(arbre n)
  {
  if (n==null) {}
  else
    {
    System.out.print(n.lettre);
    if ((n.gauche!=null)&&(n.droite!=null))
      System.out.print("(");
    imprime_texte_arbre(n.gauche);
    if ((n.gauche!=null)&&(n.droite!=null))
      System.out.print(",");
    imprime_texte_arbre(n.droite);
    if ((n.gauche!=null)&&(n.droite!=null))
      System.out.print(")");
    }
  }
```

Cette méthode affiche l'arbre sur une seule ligne, en imprimant les nœuds pères en premier.

```
public static void imprime_texte_arbre_bis(arbre n)
  {
  if (n==null) System.out.print("()");
  else
    {
    System.out.print(n.lettre);
    System.out.print("(");
    imprime_texte_arbre_bis(n.gauche);
    System.out.print(",");
    imprime_texte_arbre_bis(n.droite);
    System.out.print(")");
    }
  }
```

Cette méthode fait la même chose que la précédente, sauf qu'elle ajoute des () pour chaque feuille : *a (() , ())* est une feuille.

3. Décodage et encodage

Les méthodes suivantes vont servir à coder une chaîne dans une séquence ou à décoder une séquence pour obtenir une chaîne de caractères, le tout en se basant sur l'arbre de Huffman que l'on construira par la suite. Pour ce faire, on crée l'objet *séquence* avec ses méthodes et constructeurs.

a. Constructeurs du type séquence

```
sequence()
  {
  donnee=0;
  suivant=null;
  }

sequence(int z)
  {
  donnee=z;
  suivant=null;
  }

sequence(sequence s)
  {
  donnee=s.donnee;
  suivant=s.suivant;
  }
```

Avec de tels constructeurs, on va pouvoir créer une séquence à un élément 0 *(sequence())*, une séquence à un élément binaire 0 ou 1 *(sequence(int z))*, et effectuer une copie de séquence *(sequence(sequence s))*.

b. Fonction decode_un

```
public static sequence decode_un(arbre h,sequence b)
 {
 if ((b.suivant==null)&&!(h.estfeuille(h)))
  {
  if (b.donnee==1)
    {
    System.out.println(h.gauche.lettre);
    return(null);
    }
  else
    {
    System.out.println(h.droite.lettre);
    return(null);
    }
  }
 else
  {
  if(h.estfeuille(h))
    {
    System.out.print(h.lettre);
    return(b);
    }
  else
    {
    if (b.donnee==1)
      {
      return(decode_un(h.gauche,b.suivant));
      }
    else
      {
      return(decode_un(h.droite,b.suivant));
      }
    }
  }
 }
```

Cette méthode affiche l'élément codé en tête d'une séquence b et retourne le reste de la séquence binaire, le tout suivant l'arbre de Huffman.

c. Procédure decode

```
public static void decode(arbre h,sequence b)
 {
 if (b==null) {}
 else
  {
  decode(h,decode_un(h,b));
  }
 }
```

Cette méthode fait simplement appel de manière récursive à la méthode *decode_un* pour décoder toute la séquence en caractères.

d. Fonction code_un

```
public static boolean code_un(arbre h,char c)
 {
 if (h==null) return(false);
 else
  {
  if (h.lettre==c) return(true);
  else
   {
   if (appartient(h.droite,c))
    {
    s=s+0;
    return (false|code_un(h.droite,c));
    }
   else
    {
    if (appartient(h.gauche,c))
     {
     s=s+1;
     return(false|code_un(h.gauche,c));
     }
     else return(false);
    }
   }
  }
 }
```

A partir d'un caractère et d'un arbre de Huffman, cette méthode ajoute à une séquence le code binaire correspondant au caractère dans l'arbre ; elle renvoie *FAUX* si le caractère n'est pas dans l'arbre. Cette méthode fait appel à la méthode *appartient* qui vérifie si le caractère se trouve dans l'arbre :

```
public static boolean appartient(arbre h, char c)
 {
 if (h==null) return(false);
 else
  {
  if (h.lettre==c) return(true);
  else
   {
   if (appartient(h.gauche,c))
    {
    return (false|appartient(h.gauche,c));
    }
   else
    {
    if (appartient(h.droite,c))
     {
     return(false|appartient(h.droite,c));
     }
     else return(false);
    }
   }
  }
 }
```

e. Fonction code

```
public static sequence code(arbre h,String c)
   {
   sequence seq=new sequence();
   sequence seq2=new sequence();
   for (int i=0;i<c.length();i++)
     {
     s="";
     code_un(h,c.charAt(i));
     seq2=conversion(s);
     ajouter_queue_bis(seq2,seq);
     }
   return(seq.suivant);
   }
```

La manipulation d'une séquence nécessitant de connaître la taille totale de celle-ci, il a fallu utiliser une chaîne puis la convertir, c'est pourquoi on fait appel à la méthode *conversion*. De plus, il faut pouvoir concaténer deux séquences ; c'est le rôle de la méthode *ajouter_queue_bis*, tandis que la méthode *ajouter_queue* n'ajoute qu'un seul élément en fin de séquence :

```
public static sequence conversion(String s)
  {
  char[] tab=new char[256];
  tab=s.toCharArray();
  sequence x=new sequence();
  for (int i=0;i<s.length();i++)
    {
    ajouter_queue(x,Integer.parseInt(""+tab[i]));
    }
  return(x.suivant);
  }
```

```
public static sequence ajouter_queue(sequence t,int d)
  {
  sequence taux=t;
  sequence tmp=new sequence(d);
  while (taux.suivant!=null)
    {
    taux=taux.suivant;
    }
  taux.suivant=tmp;
  return(t);
  }
```

```
public static sequence ajouter_queue_bis(sequence t,sequence u)
  {
  sequence taux =u;
  while (taux.suivant!=null)
    {
    taux=taux.suivant;
    }
```

```
  taux.suivant=t;
  return(u);
  }
```

Par ces différentes méthodes, il est maintenant possible de coder/décoder une chaîne de caractères/séquence à partir d'un arbre de Huffman qu'il reste à construire.

4. Construction de l'arbre de codage

Pour que le codage de Huffman soit efficace, il faut que l'arbre de départ soit bien choisi. Pour les caractères les plus redondants dans une chaîne, il faut le code le plus court possible ; ainsi de tels caractères seront placés près du sommet de l'arbre. Inversement, les caractères les moins répandus seront placés en bas de l'arbre.

On va donc créer le type *arbrepoids* composé d'un *élément*, l'arbre correspondant au caractère, et d'un *poids*, la fréquence d'apparition du caractère dans le texte. Le type *liste*, quant à lui, se charge de ranger les *arbrepoids* par ordre croissant de poids de caractère ; c'est à partir de cette liste que l'arbre sera construit.

a. Constructeurs du type liste

Le type *liste* est composé d'objets *arbrepoids* ; il faut d'abord définir le type :

```
public class arbrepoids
  {
  arbre element;
  int poids;

  arbrepoids(arbrepoids a)
    {
    element=a.element;
    poids=a.poids;
    }

  arbrepoids(arbre a,int p)
    {
    element=a;
    poids=p;
    }
  }
```

Les méthodes habituelles sont implémentées, telles que la création ou la copie d'un *arbrepoids*.

```
public static liste inserer(arbrepoids p,liste l)
  {
  if(l==null) return  new liste(p,null);
  if(l.donnee!=null)
    {
    if(p.poids<=l.donnee.poids) return(new liste(p,l));
```

```
    else
      {
      return fusion(new liste(p,null),l);
      }
    }
  else return new liste(p,null);
  }
```

Cette méthode permet d'insérer un élément *arbrepoids* à sa place dans une liste triée par poids croissants de caractères. Elle fait appel à la méthode *fusion* qui concatène deux listes :

```
public static liste fusion(liste a,liste b)
  {
  if(a==null) return b;
  a.suivant=fusion(a.suivant,b);
  return a;
  }
```

Pour obtenir une liste triée, on utilise la méthode *tri_bulle* suivante :

```
public void tri_bulle()
  {
  liste laux;
  if ((this==null)||(this.suivant==null)||(this.donnee==null));
  else
    {
    laux=this;
    int test=0,i;
    arbrepoids tmp;
    while (laux.suivant!=null)
      {
      if ((laux.suivant).donnee.poids<laux.donnee.poids)
        {
        tmp=(laux.suivant).donnee;
        (laux.suivant).donnee=laux.donnee;
        laux.donnee=tmp;
        test=1;
        }
      laux=laux.suivant;
      }
    if (test==0);
    else
      {
      test=0;
      this.tri_bulle();
      }
    }
  }
```

Enfin, la méthode *supprimer_tete* permet de supprimer un élément de poids le plus faible, donc la tête de liste :

```
public static liste supprimer_tete(liste l)
  {
  if (l==null) return(null);
  else return(l.suivant);
  }
```

b. Fonction make_huff

```
public static arbre make_huff(liste l)
  {
  if(l==null)
    {
    System.out.print("Liste vide \n");
    return null;
    }
  else
    {
    if(l.suivant==null) return l.donnee.element;
    else
      {
      arbre a=new arbre(tete(l),tete(l.suivant),'*');
      arbrepoids p= new arbrepoids(a,l.donnee.poids+l.suivant.donnee.poids);
      l=l.suivant.suivant;
      return (make_huff(new liste(p,l)));
      }
    }
  }
```

Cette méthode permet, à partir d'une liste triée correctement, de construire l'arbre de Huffman ; celui-ci est construit dans l'ordre de la liste, c'est-à-dire de bas en haut (faible poids → poids élevé). Elle fait appel à la méthode *tete* qui renvoie la tête d'une liste :

```
public static arbre tete(liste l)
  {
  if (l==null) return(null);
  else return(l.donnee.element);
  }
```

c. Fonction make_liste

```
public static liste make_liste(String ch1)
  {
  int i;
  int[] freq=new int[256];
  for(i=0;i<ch1.length();i++)
    {
    if((int)ch1.charAt(i)>256)
      System.out.print("caractère "+ch1.charAt(i)+" non valide");
    else
```

```
    freq[(int)ch1.charAt(i)]++;
   }
   liste l=new liste();
   for(i=0;i<256;i++)
    {
    if(freq[i]!=0)
     {
     arbre a=new arbre(null,null,(char)i);
     arbrepoids ab=new arbrepoids(a,freq[i]);
     l=inserer(ab,l);
     }
    }
 return l;
 }
```

Cette méthode est très importante ; elle permet de construire la liste qui contiendra les caractères et leur fréquence d'apparition pour pouvoir ensuite créer l'arbre de codage optimisé.

Fonctions annexes et problèmes

D'autres fonctions sont présentes dans les différentes classes utilisées, notamment pour l'affichage de certains éléments (déboguage).

Normalement, avec les classes définies, en entrant une chaîne de caractères, on devrait avoir l'arbre correct puis la séquence correspondant au codage de la chaîne suivant l'arbre puis de nouveau la chaîne décodée, pour vérification. Malheureusement, un problème dans le tri de la liste des poids fait qu'il manque un élément de la liste (celui de la fin) et qu'il y en a un de trop au début, d'où une création erronée de l'arbre qui ne respecte pas les poids des lettres de la chaîne de caractères.

Par exemple, pour le texte « aababbbcccbbbbb » où apparaissent trois fois la lettre a, neuf fois la lettre b et trois fois la lettre c, les poids de la liste non triée sont 3 9 3 ; en la triant on obtient 0 3 3 au lieu de 3 3 9. Il en résulte un arbre puis une séquence tous deux faux, bien que l'on arrive à décoder le bon texte en sortie, les méthodes d'encodage/décodage étant correctes.

Conclusion

Ce projet fut assez intéressant à mener puisqu'il m'a permis de comprendre le fonctionnement de Java et plus généralement de la programmation objet. Le problème qui apparaît lors de la création de liste n'a pas pu être situé précisément ni identifié à ce jour ; c'est dommage car toutes les autres méthodes fonctionnent, et à cause d'une seule qui est incorrecte, tout le programme fonctionne mal.

Faute de temps, l'encodage/décodage du fichier ainsi que le codage dynamique n'ont pas pu être réalisés mais l'important était de bien s'initier au langage Java, sachant que mon stage ST40 en septembre portera justement sur ce langage. Enfin, ayant précédemment utilisé le codage/décodage de Huffman manuellement dans le cadre d'exercices (IN42), il est intéressant de voir comment une théorie peut être informatisée pour un traitement rapide et systématique de données.