Teoría de la Información y Codificación

Práctica 1: Creación código libre de prefijos con árboles binarios y cálculo de Kraft-Millan

José A. Montenegro Montes

26 de septiembre de 2014

1. Enunciado

La práctica se centra en la construccion de códigos libre de prefijo utilizando un árbol, en este caso binario. El usuario pasará los parámetros del código deseado y obtendremos una codificación si es posible.

El alumno deberá realizar el método que realiza el cálculo de Kraft-McMillan (KM) descrito en las transparencias del tema 2 (pagina 24). Verifique los resultados con el Ejercicio 7 (pagina 29) de las transparencias.

El número KM nos permite verificar si es posible crear un código libre de prefijos según unos parámetros datos.

El objetivo principal que persigue la práctica es observar que mediante el calculo de Kraft-McMillan sabemos de antemano si se puede o no construir el código con los parametros seleccionados, sin tener que calcularlos. Observaremos el tiempo que tarda realizar el calculo con el arbol y el tiempo en calcular con Kraft-McMillan, tal y como muestra la figura 1.

Tratando el tema de optimización observaremos que mediante una pequeña modificacion en el calculo de Kraft-McMillan (utilizando una alternativa a Math.pow) podemos bajar notablemente el tiempo empleado en su cálculo (figura 2).

2. Conclusiones

El objetivo de esta práctica es desarrollar código libres de prefijo mediante recorrido en un árbol binario, estructura que le resultará familiar al alumno.

Observaremos la utilidad de aplicar la fórmula de Kraft-McMillan para hacer más eficiente nuestro algoritmo.

Finalmente, veremos que aunque en la mayoría de los casos, es preferible utilizar la librería estándar de Java, en ocasiones puntuales, una implementación propietaria puede mejorar los tiempos .

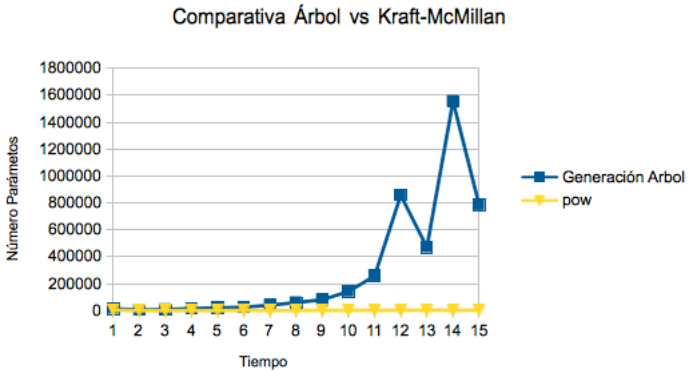

Figura 1: Generación código Árbol vs Kraft-McMillan

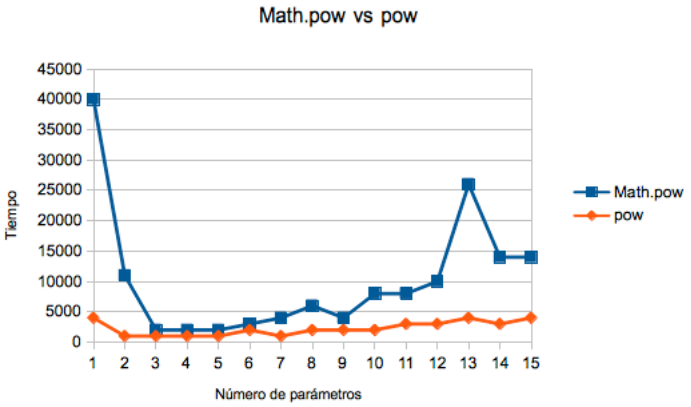

Figura 2: Utilización Math.pow vs pow en Kraft-McMillan

3. Código

Clase BinaryTreeCod

```
/*****************************************
 * Practica 1 Teoria de la Informacion y Codificacion. (Acentos eliminados)
 *
 * La practica muestra la construccion de codigos libre de prefijo utilizando un
 * arbol, en este caso binario.
 * La construcion es realizada basado en los parametros por nivel.
 *
 * Tal y como detallamos en las transparencias del tema 2 (pagina 24) realizamos
 * el calculo de Kraft-McMillan.
 * Verifique los resultados con el Ejercicio 7 (pagina 29) de las transparencias.
 *
 * El objetivo principal que persigue la practica es observar que mediante el
 * calculo de Kraft-McMillan sabemos de antemano si se puede o no construir el
 * codigo con los parametros seleccionados.
 *
 * Observaremos el tiempo que tarda realizar el calculo con el arbol y el tiempo
 * en calcular con Kraft-McMillan.
 *
 * Ademas observaremos que realizando una modificacion en el calculo de
 * Kraft-McMillan, utilizando una
 * alternativa a Math.pow.
 *
 *  @author joseamontenegromontes
 *
 *  Version 1: 4 Octubre 2013
 *
 */
package Practica1;

import java.util.ArrayList;
import java.util.List;



public class BinaryTreeCod {


        Node root=new Node("");
        int levelTree=1;

/*****************************************
```

```
42  * Clase Privada Node.
43  *
44  * Almacena los datos de los nodos del arbol.
45  * @author joseamontenegromontes
46  *
47  */
48         private class Node {
49
50                 String name;
51                 Boolean used=false;
52
53                 Node leftChild=null;
54                 Node rightChild=null;
55
56 /*******
57  * Constructor
58  * @param name
59  */
60                 Node(String name) {
61
62                         this.name = name;
63
64                 }
65 /************
66  * Imprime los valores del nodo. Para depuracion.
67  */
68                 public String toString() {
69                         if (name.isEmpty()) return "raiz";
70                         else return name +" = "+used;
71                 }
72 /***************
73  *         Establece el nodo de la derecha como utilizado
74  */
75                 public boolean setRUsed (){
76
77                         return setUsed (rightChild);
78
79                 }
80
81 /***************
82  *         Establece el nodo de la izquierda como utilizado
83  */
84
85                 public boolean setLUsed (){
86
87                         return setUsed (leftChild);
```

```
88
89                  }
90
91  /***************
92   *          Auxiliar para setLUsed y setRUsed
93   */
94                  private boolean setUsed (Node node){
95                          boolean returnValue=true;
96
97                          if (node.isUsed()) returnValue=false;
98                          else          node.used=true;
99
100                         return returnValue;
101                 }
102
103
104 /***************
105  * Conocemos si un nodo esta utilizado
106  */
107
108                 public boolean isUsed (){
109                         return used;
110                 }
111
112         }
113 /******************** Fin clase privada Node  **********/
114
115
116 /*****************************
117  * Constructor
118  * @param levelp niveles del arbol
119  */
120         public BinaryTreeCod (int levelp){
121                 levelTree=levelp;
122                 generate(root,levelTree);
123         }
124
125
126 /*****************************
127  * Auxiliar para crear el arbol por niveles
128  * @param focusNode
129  * @param level
130  */
131
132         private void generate(Node focusNode,int level) {
133
```

```
134                 if (level != 0) {
135 
136                         String izq=focusNode.name.concat("0");
137                         String dch=focusNode.name.concat("1");
138 
139                         focusNode.leftChild = new Node(izq);
140                         focusNode.rightChild = new Node(dch);
141 
142                         generate(focusNode.leftChild,level-1);
143                         generate(focusNode.rightChild,level-1);
144 
145                 }
146 
147         }
148 
149 
150 
151 /***************
152  * Construimos un codigo libre de prefijos segun los parametros
153  * @param n
154  * @throws TreeException
155  */
156 
157 public boolean buildCodeParameters(int []n) throws TreeException {
158         int size= n.length;
159         boolean result=true;
160 
161         if (size > levelTree ) throw new TreeException ("Not enough levels!!");
162 
163         for (int i=0;i<size;i++){
164                 if(n[i]>0)        result=levelFreePrefix(i+1, n[i]); //Eliminate  unnecessa
165                 if (!result) return false;
166         }
167         return true;
168 }
169 
170 /**********
171  *  Auxiliar de buildCodeParameters. Rellena los n elementos de un nivel.
172  * @param level
173  * @param n
174  */
175 private boolean levelFreePrefix(int level, int n) {
176         int used=levelAuxPrefixFree(root,level, n);
177 
178         if (used==0) return true;
179         else                 return false;
```

```
180 }
181
182
183 /*****************************
184  *          AUXILIAR
185  *
186  * Auxiliar de levelFreePrefix. Rellena usedLevel elementos de un nivel dado level, desde no
187  * @param focusNode
188  */
189 private int levelAuxPrefixFree(Node focusNode,int level, int usedLevel) {
190
191  if (level==1){
192     if (usedLevel>0 & !focusNode.isUsed())        if (focusNode.setLUsed()) usedLevel--; //l
193     if (usedLevel>0 & !focusNode.isUsed())        if (focusNode.setRUsed())
194         usedLevel--;
195
196         }
197  else{
198          if (usedLevel>0 & !focusNode.leftChild.isUsed() ){
199                 usedLevel=levelAuxPrefixFree(focusNode.leftChild,level-1, usedLevel);
200          }
201         if (usedLevel>0 & !focusNode.rightChild.isUsed() ) {
202           usedLevel=levelAuxPrefixFree(focusNode.rightChild,level-1, usedLevel);
203          }
204         }
205
206         return usedLevel;
207 }
208
209
210 /*************
211  * Imprime el arbol generado
212  */
213         public void printBinaryTree(){
214
215                  printBinaryTree(root, 0);
216
217         }
218
219 /************
220  *          Auxiliar de printBinaryTree
221  *
222  * @param root
223  * @param level
224  */
225
```

```
226         private static void printBinaryTree(Node root, int level){
227            String output=null;
228 
229                 if(root==null) return;
230 
231                 printBinaryTree(root.leftChild, level+1);
232 
233 
234             if(level!=0){
235                 for(int i=0;i<level-1;i++)
236                         System.out.print("|\t");
237 
238                 if (root.isUsed()) output=root.name;
239                 else output="x";
240                 System.out.println("|-------"+output);
241 
242             }
243             else
244                if (root.isUsed()) System.out.println(root.name);
245                else                            System.out.println("x");
246 
247 
248             printBinaryTree(root.rightChild, level+1);
249         }
250 
251 
252 /***********
253  * Verifica si el arbol es libre de prefijo
254  */
255         public boolean prefixFree(){
256 
257                 return prefixFreeAux(root,false);
258 
259 
260         }
261 /*************
262  * Auxiliar de prefixFree
263  * @param node
264  * @param checked
265  * @return
266  */
267         public boolean prefixFreeAux(Node node,boolean checked){
268                 boolean check=false;
269                 boolean end=false;
270 
271                 if (node==null) return true;
```

```
272
273                 if (!node.name.isEmpty()) { //Eliminate root node.
274                         check=node.isUsed();
275
276                         if (check && checked) {
277                                 return false; //Non prefix Free.
278                         }
279                         else {
280                                 checked=check|checked;  // Super or working node selected
281                         }
282                 }
283
284                 end=prefixFreeAux(node.leftChild,checked);
285                 if (end) end=prefixFreeAux(node.rightChild,checked);
286
287                 return end;
288 }
289
290 /*********************
291  *          Calculo de la potencia binaria. 2^x
292  *
293  * @param x
294  * @return
295  */
296
297 public double pow (int x){
298
299         return (1 << x);
300 }
301
302
303 /*********************
304  * Calculo de kraftMcMillan segun parametros del codigo usando Math.pow
305  * @param ni
306  * @return
307  */
308 public double kraftMcMillan(int [] ni) {
309
310                 double K = 0;
311                 //ToDO: Tu codigo aqui
312                 return K;
313         }
314
315 /************************
316  * Calculo de kraftMcMillan segun parametros del codigo usando nuestro pow
317  * @param ni
```

```
318  * @return
319  */
320 public double kraftMcMillanOptimization(int [] ni) {
321         double K = 0;
322         //ToDO: Tu codigo aqui
323         return K;
324 }
325 
326 
327 /**********
328  * Imprime los parametros generados.
329  * ToDo: parameters puede ser eliminados para no evitar confusiones. Deben ser los mismos qu
330  * para construir el arbol. Se pueden almacenar como atributo de la clase.
331  * @param parameters
332  */
333 
334 public void  printCodes(int [] parameters) {
335         List <String> CodeList=getCodes();
336         int size=CodeList.size();
337         int sizep=parameters.length;
338 
339         System.out.print("\nCode obtained with ");
340 
341         for (int i = 0; i < sizep; i++) {
342                 System.out.print(parameters[i]+" ");
343         }
344 
345         System.out.println(" parameters: ");
346 
347         for (int i = 0; i < size; i++) {
348                 System.out.print(" "+CodeList.get(i)+" ");
349         }
350 }
351 
352 /*******************
353  * Obtiene los codigos generados.
354  * @param ni
355  * @return
356  */
357 public List <String>  getCodes() {
358         List <String> CodeList=new ArrayList<String>();
359 
360         CodeList= inOrderTraverseTree(root,CodeList);
361 
362         return CodeList;
363 }
```

```
364
365
366 /*******************
367  * Auxiliar getCodes. Recorrido en orden, almacena solamente los nodos seleccionados.
368  * @param ni
369  * @return
370  */
371 private List <String> inOrderTraverseTree(Node focusNode,List <String> code) {
372
373         if (focusNode != null) {
374                 if (focusNode.isUsed()) code.add(focusNode.name);
375                 code=inOrderTraverseTree(focusNode.leftChild,code);
376                 code=inOrderTraverseTree(focusNode.rightChild,code);
377                 return code;
378         }
379
380         return code;
381 }
382
383
384 /**************
385  * Funcion para verificar la practica.
386  * @param parameters. Crear codigo segun unos parametros.
387  * @param printTree. Decido si quiero imprimir o no el arbol.
388  */
389 static public void Test (int[]parameters,boolean printTree){
390
391  boolean result=true;
392  double KMvalue=0;
393  long ini,fin; //Time variables.
394
395  int leng= parameters.length;
396  BinaryTreeCod code = new BinaryTreeCod(leng);
397
398
399  try {
400         ini=System.nanoTime();
401         result=code.buildCodeParameters(parameters);
402         fin=System.nanoTime();
403
404
405         if(result){
406          System.out.println("BinaryTreeMethod:Found prefix free code in "+(fin-ini)+ " nanos
407
408         if (printTree) code.printBinaryTree(); //Print tree
409
```

```
410         ini=System.nanoTime();
411                 boolean isPrefixFree=code.prefixFree();
412         fin=System.nanoTime();
413 
414         System.out.println("Verifying prefix free: "+isPrefixFree+" in "+(fin-ini)+ " nanose
415 
416         }
417         else
418         System.out.println("BinaryTreeMethod: I can not generate prefix free code usign this
419 
420 
421         } catch (TreeException e) {
422                 // TODO Auto-generated catch block
423                 e.printStackTrace();
424         }
425 
426          ini=System.nanoTime();
427                 KMvalue = code.kraftMcMillan(parameters);
428          fin=System.nanoTime();
429 
430         System.out.println("kraft- McMillan Regular      Value: "+KMvalue+" in "+(fin-ini)+"
431 
432          ini=System.nanoTime();
433                 KMvalue = code.kraftMcMillanOptimization(parameters);
434          fin=System.nanoTime();
435 
436         System.out.println("kraft- McMillan Optimization Value: "+KMvalue+" in "+(fin-ini)+"
437 
438         code.printCodes(parameters);
439 
440         System.out.println("");
441 }
442 /*****************************************
443  *
444  * @param args
445  */
446 
447         public static void main(String[] args) {
448 
449                 boolean printTree=true;
450 
451                 int [] parametersC1={0,1,4,3};
452                 BinaryTreeCod.Test(parametersC1,printTree);
453         }
454 }
455 
```

Clase TreeException Java.

```java
package Practica1;

public class TreeException extends Exception {

          public TreeException(String message){
             super(message);
          }

}
```