Dictionary Methods

CS6025 Data Encoding

Yizong Cheng

3-26-15

Phrases

- Partition the input string of characters into non-overlapping substrings called *phrases*.
- Replace each phrase with a bit string called a *codeword*.
- The mapping between phrases and codewords is given in a *dictionary*.

Textual Substitution

- fixed-length phrases
- variable-length phrases
- fixed-length codewords
- variable-length codewords
- replace more frequent phrases with shorter codewords

Static Dictionary

- static dictionary
- can be precomputed for each input string
- must be considered as part of the compressed version of the input

Dynamic Dictionary

- Dictionary only consists of substrings of the input string.
- Codewords can be viewed as pointers to another occurrence of the same substring in the input string.
- unidirectional or bidirectional
- Single-pass unidirectional dynamic dictionary is the most popular.

LZ77

- White 1967 Proceedings of the IEEE paper on dictionary encoding.
- Ziv and Lempel 1977 Trans. IT paper “A universal algorithm for sequential data compression” (LZ77).
- They proved that for ergodic sources, LZ77 is asymptotically optimal to be arbitrarily close to the entropy of the source.

LZ77 Codewords

- a pointer to the start location of the last occurrence of the phrase
- the length of the phrase
- the first character of the uncompressed portion of the input
- skip this character for next phrase
- dictionary is suffix complete (all suffixes of a phrase are phrases).

gzip

- a LZ77 variant
- uses a hash table to store the location of every occurrence of each 3-character phrases
- Huffman coding the offset and length entries of each codeword
- distributed for GNU by the Free Software Foundation
- Implemented in java.util.zip

Others using LZ77

- winzip
- PKZIP
- PNG
- arj
- LHarc

LZ78

- Ziv and Lempel 1978 Trans IT paper “Compression of individual sequences via variable-rate coding”
- Welch 1984 U.S. Patent (LZW)
- achieves entropy faster than LZ77
- UNIX compress
- GIF
- modem compression standards v.42bis and v.44

LZW

- dictionary initially consists only of single-character substrings
- After each parsing step, the phrase parsed is concatenated with the first character of the uncompressed portion of the input and inserted in the dictionary as a new phrase
- dictionary is prefix complete

LZW Dictionary

- Each phrase in the dictionary is stored as its last character and the index of its prefix.
- Dictionary is dynamically generated by both encoder and decoder.
- Dictionary is implemented as a 2-column table (parallel arrays) for decoder but a tree for encoder.
- The index of a phrase in the dictionary is the codeword of the phrase.

LZW, Adaptive Dictionary

- The first 256 entries of the dictionary are the 256 single-byte phrases.
- Each new entry is for a multiple-byte phrase, whose prefixes are already entries of the dictionary.
- Always find the longest matched phrase in the dictionary and add a new entry one byte longer.
- The file is partitioned into phrases.
- The file is encoded as a sequence of the codewords for these phrases.

Example 1

- alf_eats_alfalfa
- 97 (a), new 256 (al), (97,108)
- 108 (l), new 257 (lf), (108,102)
- 102 (f), new 258 (f_), (102,32)
- 32 (_), new 259 (_e), (32,101)
- 101 (e), new 260 (ea), (101,97)
- 97 (a), new 261 (at), (97,116)

Example 1

- alf_eats_alfalfa
- 116 (t), new 262 (ts), (116,115)
- 115 (s), new 263 (s_), (115,32)
- 32 (_), new 264 (_a), (32,97)
- 256 (al), new 265 (alf), (256,102)
- 102 (f), new 266 (fa), (102,97)
- 265 (alf), new 267 (alfa), (265,97)

Table Lookup: Encoder

- Find table index for the longest matching phrase.
- Data structure for table: a tree.
- Each node corresponds to a phrase in the dictionary.
- Each branch marked with a byte/character.
- Longest match found when no branch to go.

Table Lookup: Decoder

- From index/codeword to phrase.
- A phrase is represented as (codeword/index to the prefix, last byte)
- Data structure: 2 arrays, one for pointer to prefix, one for last byte.
- Given an index, trace pointers back to a single byte phrase (<256).
- Output the phrase as the reverse of the trace.
  - Reverse printout of a linked list using recursion in outputPhrase()

Table Building: Encoder

- Each time a match is found, a new branch and leaf is added to the tree.
- Implementation using a binary tree.
- Left child represents oldest child.
- Right child represents next sibling.

```
class Node{  // for building the dictionary tree
  Node nextSibling;
  Node children;
  int symbol;
  int code;

  public Node(Node n, Node c, int s, int d){
    nextSibling = n; children = c; symbol = s; code = d;
  }

  Node findChild(int s){
    Node k = children; for (; k != null; k = k.nextSibling)
    if (s == k.symbol) break;
    return k;
  }

  void addChild(int s, int d){
    Node newChild = new Node(children, null, s, d);
    children = newChild;
  }
}
```

The LZW Encoding Program

```
public class H18A{

 static final int bufSize = 65536;
 static final int dictionaryCapacity = 4096;
 byte[] buffer = new byte[bufSize];
 int dictionarySize = 0;
 Node dictionary = null;
 int dataLength = 0;
 int dataPosition = 0;

 public static void main(String[] args){
   H18A h18 = new H18A();
   h18.initializeDictionary();
   h18.readBlock();
   h18.encodeBlock();
 }
}
```

```
void readBlock(){
   try {
     dataLength = System.in.read(buffer);
   } catch (IOException e){
     System.err.println("IOException");
     System.exit(1);
   }
 }

 void initializeDictionary(){
   dictionary = new Node(null, null, -1, -1);
   for (int i = 0; i < 256; i++)
      dictionary.addChild(i, i);
   dictionarySize = 256;
   dataPosition = 0;
 }
```

```
int longestMatch(){
  int codeword = -1;
  Node node = dictionary;
  Node parentNode = null;
  while (dataPosition < dataLength && node != null){
    parentNode = node;
    node = node.findChild(buffer[dataPosition++]);
  }
  if (node == null){
    dataPosition--;
    codeword = parentNode.code;
    if (dictionarySize < dictionaryCapacity)
      parentNode.addChild(buffer[dataPosition], dictionarySize++);
  }else codeword = node.code;
  return codeword;
}

void encodeBlock(){
  while (dataPosition < dataLength){
    int codeword = longestMatch();
    if (codeword >= 0) System.out.println(codeword);
  }
}
```

Table Building: Decoder

- Encoded data is a sequence of codewords/table-indexes.
- For each codeword, build a new entry and place the codeword/index as its prefix component.
- The second component is the first byte of the phrase represented by the **next** codeword in the encoded data.
- Loop: read next codeword, output the corresponding phrase in the dictionary using the codeword as the index, fill the second component of the last phrase with the first byte of the phrase just output, fill the first component of the next entry with the codeword.

A Special Case in Decoder

- alf_eats_alfalfalfa
- 256 (al), new 265 (alf), (256,102)
- 102 (f), new 266 (fa), (102,97)
- 265 (alf), new 267 (alfa), (265,97)
- 267 (alfa)
- The last phrase (alfa) cannot be output before ‘a’ is filled.
- Condition: codeword == dictionarySize – 1, the second component (last byte) of the last phrase in the dictionary has not been filled.
- Solution: this phrase must have its last byte equal to its first byte.

```
public class H18B{

 static final int dictionaryCapacity = 4096;
 int[] prefix = new int[dictionaryCapacity];
 int[] lastSymbol = new int[dictionaryCapacity];
 int dictionarySize = 0;
 int firstSymbol = 0;

 void initializeDictionary(){
   for (int i = 0; i < 256; i++){
      prefix[i] = -1; lastSymbol[i] = i;
   }
   dictionarySize = 256;
 }

 public static void main(String[] args){
   H18B h18 = new H18B();
   h18.initializeDictionary();
   h18.decode();
 }
}
```

```
void outputPhrase(int index){  // recursive
  if (index >= 0 && index < dictionarySize){
    outputPhrase(prefix[index]);
    System.out.write(lastSymbol[index]);
    if (index < 256) firstSymbol = lastSymbol[index];
  }
}

void decode(){
  Scanner in = new Scanner(System.in);
  while (in.hasNextLine()){
    int codeword = Integer.parseInt(in.nextLine());
    outputPhrase(codeword);
    if (dictionarySize < dictionaryCapacity){
      lastSymbol[dictionarySize - 1] = ?
      prefix[dictionarySize++] = ?
    }
  }
}
```

Homework 18: due 3-31-15

- Complete H18B.java as the inverse of H18A.java.
- Decode test18.txt using H18B and return the result along with the source code.