CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 19: Lab #13

Fall 2014
MWF 9:05-9:55
Vickery 100

1 LZ78 Text Compression Using a Trie

In this lab we will write a simplified variant of the Lempel-Ziv 78 text compression algorithm, while gaining practice using the *trie* data structure (pronounced "try"). Recall that a trie encodes a set of strings "vertically" in all of its root-to-leaf paths. Since a trie is not necessarily a binary tree (its nodes can have many children), each node typically maintains a pointer to its first child and its next sibling. This way, we can enumerate the children of a node by stepping to its first child and then following the linked list of next sibling pointers.

2 Review: LZ78 Compression

The main idea behind LZ78 compression is similar to most other compression protocols: replace repeated instances of the same substring with shorter references to their first occurrence. We illustrate how this process works using a trie on an example.

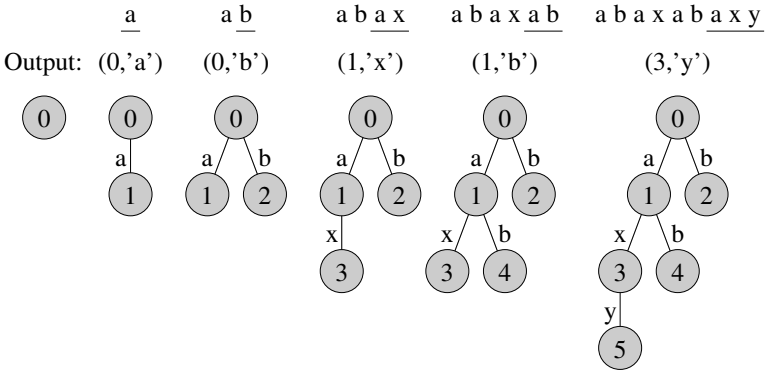

Figure 1: Outputting a series of pairs that specifies how to encode the text "abaxabaxy". Along the way, we build up a trie. The steps of the process are illustrated from left to right.

Consider the string "abaxabaxy". We start with an empty trie consisting of just a root node. Each node has an integer ID (the root has ID zero), and we assign IDs sequentially to the nodes we add to the trie.

We now scan through our string, building a trie in the process and also writing out the compressed representation of our text as a sequence of pairs. Each pair (i, c) tells us how to add a new node to the trie – by adding the next node as a child of the node with id i, labeled with character c.

As shown in Figure 1, after we see the characters 'a' and 'b', these both get added as children of the root. We indicate this by outputting $(0,'a')$ and $(0,'b')$. We then try to encode the next part of our string, starting with 'a', which already exists in the trie, so we scan down the trie while reading through the string until we find a mismatch (the 'x'), causing a new branch to be added. The process continues until all the characters in the text are read in (we treat the EOF at the end of the input as the last character, to ensure that we always create a branch representing the last few characters in the string).

3 Encoding

To help you get started with this lab, a short encoding program is provided here:

`/group/course/cpsc212/f14/lab13/encode.cpp`

If you compile and run it, it will read from standard input and print the encoded result on standard output. For example, if you type our example string `abaxabaxy`, followed by a carriage return and then CTRL-D (which send EOF), then you will get as output:

```
0 97
0 98
1 120
1 98
3 121
0 10
0 -1
```

These are the pairs of the form (i, c) produced by the encoder. The first line tells us that we need add 'a' (ASCII value 97) as a child of node 0, and so on. The 10 near the end is the ASCII code for the newline character, and the -1 at the end represents the final EOF.

In a real compression application, the output of the encoder would be written out in a more concise way, packed into a stream of bits, instead of as a series of plain-text numbers. However, for the purposes of this exercise, we will stick with the simpler format given above.

4 Decoding

Your task in this lab is to write a decoder that will take the result of the encoder as input and produce the original text as output. By piping the output of the encoder straight into your decoder, we should obtain the original input text; for example:

```
cat > test.txt
The cows attack at dawn.
cat test.txt | ./encode | ./decode
The cows attack at dawn.
```

To write your decoder, you may want to draw inspiration from looking at the code for the encoder. However, you probably want to do a few things differently. For starters, in the trie maintained by the decoder, it may not be necessary to maintain first child and next sibling pointers, but parent pointers might be useful. In fact, if you look at the code for the encoder, you will notice that it doesn't use pointers at all – since nodes are identified with integer IDs, it stores them instead in a vector, using the integer ID of a node instead of a pointer to reference the node. You may want to do the same thing in your decoder.

To make sure it works properly, you will want to test your encoder and decoder on large files. For example:

```
cat some_large_file | ./encode | ./decode > output_file
diff some_large_file output_file
```

The `diff` command shows differences between two files. If there are no differences (as should be the case if your decoder runs correctly), it prints nothing.

If you would like to see how well an easily-compressed file can be shortened with this method, look at `moo.txt` in the lab directory – it should compress to roughly 10% of its original size, even with the somewhat wasteful encoding strategy we are using.

5 Submission and Grading

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Friday, December 5. No late submissions will be accepted.