Advanced Theory of Computation

HomeWork2 Report

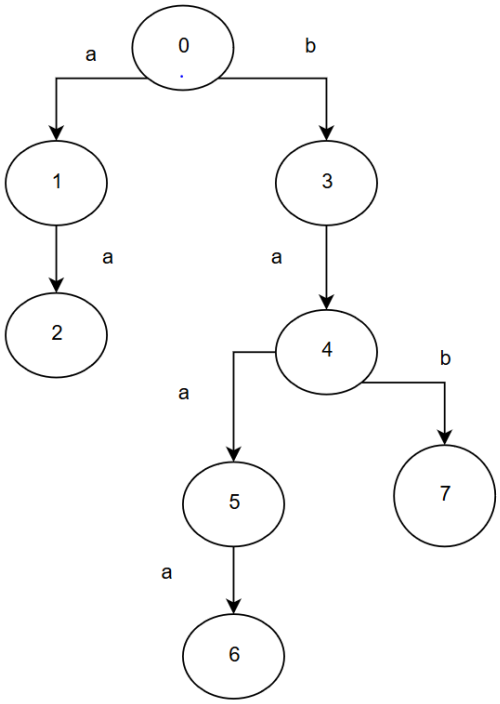
Student Name : Jeong Jae-Hwan

Student ID: 2018-26716

1. Program Structure & LZ78 Algorithm

(a) About LZ78

Brief Description : Advanced version of LZ77, there exist no such sliding windows, and lookahead buffer. Key idea is that there's no limitation for searching the longest match for entire input. In order to achieving this, we have to construct a trie(tree form, each node is called as parse number, and edges are represented as a character or string. For example, Let say there's an input such as **"aaabbabaabaaabab"**, then we use adaptive dictionary encoding techniques. We use



dictionary(instead of sliding window), and then we check longest match in input with dictionary.

for initial condition, there's only one node which denotes root node as 0, and input there's no longest match. so we output node 0(means we can't find longest match), and the next character. In this case, a. and then we add this next character "a" to node 1. Second, we know there exist longest match a in dictionary, and then we output node 1(means we can find longest match "a" in dictionary) and we encode this to 1. And then output the next character a same as first step. then we continues this steps. Finally we can get the final version of a trie, and encoded output. This case we get (0,a),

(1,a), (0,b), (3,a), (4,a), (5,a), (4,b).

Following Compression/Decompression Part's logic is as follows(same in lecture note)

- Compression
  - 1) Find the longest matching phase
  - 2) Output(phrase index, next character)
  - 3) Insert new phrase(matching phrase + next character) into dictionary.
- Decompression:
  - 1) Output a new phrase from (phrase index, next character)
  - 2) Insert the new phrase into the dictionary

(b) Implementation Details

Used edge value as child node's index(character node's ascii code) ascii code number's range is 0-127. So maximum number of child is 128. And used maximum tries' node as $< 2^{30}$ (This is related to Section 2, so refer section 2)

Files related to encoding : encoding.cpp Tries.cpp Tries.h TriesNode.cpp TriesNode.h

- encoding.cpp : LZ78 Encoding algorithm and main function
- Tries.cpp : "Tries.h" function definition
- Tries.h : Construction things about trie : make a root, and ptr(which pointers current node, if one makes a new node than, ptr points new child node, otherwise ptr points to root.(for example if matching fails).
- TriesNode.cpp : "TriesNode.h" function definition
- TriesNode.h : Construction for one TrieNode(which means parse number. In encoding phase, there's no need to save the string information that one node represents.

Files related to decoding : decoding.cpp

- decoding.cpp : LZ78 Decoding algorithm and main function
- dTries.h : Construction things about trie : this time, we have to know each node's string information and if we had to extract that information from dTries, Used vector to represent the node's string information.
- dTries.cpp : "dTries.h" function definition
- dTriesNode.cpp : "dTriesNode.h" function definition
- dTriesNode.h : Construction things about trie : each node must have their string's information. Used character array for length 1000.

2. Own idea how to do compress well

(a) method how to output parse number information

Implementation Concerns : there's a issue about how to implement the LZ78, especially for outputting to output file stream. Each step, LZ78 puts phrase number and the next character of their longest match string. The problem is that we normally always put 4 bytes to represent the node number if we use *int* variable, but actually, I tested it for example "infile.txt"(about 1.5MB) and in that case, there was about 257,045 nodes. And that means we don't have to use the all 4bytes for representing each phrase number(same as node number). In implementation limits, is that we can minimally put 1 bytes, not 1 bit. So the idea is that we can fix the limitation of representing the nodes, about $< 2^{30}$, then we use the 2 MSB bits for representing the number of the blocks(1 block means 1 byte), and the other 6bits for integers. In order to implement this idea, we have to shift the bit in *unsigned char*(*uint8_t*). cases are like $0 \sim 2^6$, $2^6 \sim 2^{14}$, $2^{14} \sim 2^{22}$, $2^{22} \sim 2^{30}$. $0 \sim 2^6$ we can use MSB 2 bits set as 00. For example, let say we use 255 (1111 1111) then we have to use 2 blocks, and that will be represented like ((0100 0000) (1111 1111)), in initial state, we check the first 2 MSB bits, 01. And we know in this case, we use 2 blocks(=2 bytes), and we can get 2 blocks for getting parse number's information. Then we can know 1111 1111 denotes the parse number's information(more details about bit shifting is implemented on File code)

3. Program development Environment

OS: Ubuntu 18.04 LTS

Kernel: Linux 4.15.0-36-generic

CPU: Intel(R) Core(TM) i7-8700k CPU@ 3.20GHz

Programming Language : C++

Compiler Version : gcc, g++ version : 7.3.0

More easy way to compile : Just stroke "make" command, then you can run it quickly

4. Measurement of LZ78 & Other compression program

Target compression program : GNU Zip (Linux base compression program)

(a) encoding time

1) : my algorithm

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./encoding CMakeCache.txt 1
0.003399
```

Run-time : 0.003399(CMakeCache.txt – my input1)

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./encoding kernel_log.txt 1
0.009713
```

Run-time : 0.009713(kernel-log.txt – my input2)

2) : GNU Zip : used shell script file to evaluate the time

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./time_check
15:02:18.307668541
  adding: CMakeCache.txt (deflated 82%)
15:02:18.312465473
```

Run-time : 0.004796932s(CMakeCache.txt – my input1)

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./time_check
15:07:51.374850870
  adding: kernel_log.txt (deflated 73%)
15:07:51.377983953
```

Run-time : 0.003133083s(kernel_log.txt – my input2)

(b) decoding time

1) My algorithm

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./decoding 1 2
0.006543
```

Run-time : 0.006543s(CMakeCache.txt – my input1)

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./decoding 1 2
0.013307
```

Run-time : 0.013307s(kernel_log.txt – my input2)

2) GNU Zip

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./time_check
15:05:16.397655870
Archive:  time_test.zip
  inflating: test/CMakeCache.txt
15:05:16.402768783
```

Run-time : 0.005112913s(CMakeCache.txt – my input1)

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ ./time_check
15:09:36.472363479
Archive:  time_test.zip
  inflating: test/kernel_log.txt
15:09:36.479311543
```

Run-time : 0.006948064s(kernel_log.txt – my input2)

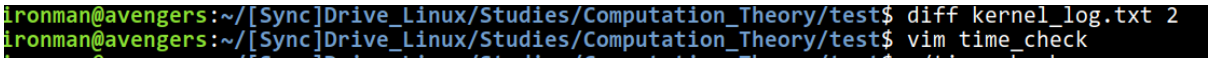
(c) check the loseless property

-> GNU Zip ensures the loseless property, in this part section, I'll show my result only.

```
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$ diff infile.txt decoding_mine
ironman@avengers:~/[Sync]Drive_Linux/Studies/Computation_Theory/test$
```

Infile.txt : Nothing occurs -> means it has the loseless property.

CMakeCache.txt(my input1) : Nothing occurs -> means it has the loseless property.



Kernel_log.txt(my input2) : Nothing occurs-> means it has the loseless property.

(d) encoded file size

1) my input1(CMakeCache.txt)

Original size : 12757Byte

Size after compressed(by my alg): 7999Byte

Size after compressed(by GNU zip): 2473Byte

2) my input2(kernel_log.txt)

Original size : 64211Byte

Size after compressed(by my alg): 40234Byte

Size after compressed(by GNU zip): 17701Byte

5. Result Analysis

(a) encoding time

GNU Zip > My algorithm

(b) decoding time

GNU Zip > My algorithm

(c) encoded file size

1)Compression ratio

1-1) input1 : mine(37%) < GNU Zip(78%)

1-2) input2 : mine(37%) < GNU Zip(72%)

(d) Correctness(satisfying Loseless Propety)

-> Both keeps Correctness for 2 input files.

(e) Analysis

It turns out that GNU Zip works way much better than my algorithm. This is because GNU Zip uses DEFLATE algorithm which is combination of LZ77 and Huffman coding. It was intended as a replacement for LZW and other patent-encumbered data compression algorithms which, at the time, limited the usability of compress and other popular archivers. It is much faster and better in the light of compression ratio than me.

6. Reference

[1] *Ziv, Jacob, and Abraham Lempel. "Compression of individual sequences via variable-rate coding." IEEE transactions on Information Theory 24.5 (1978): 530-536.*

[2] *Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." IEEE Transactions on information theory 23.3 (1977): 337-343*

[3] *http://ethw.org/Milestones:Lempel- Ziv_Data_Compression_Algorithm,_1977*

[4] *https://en.wikipedia.org/wiki/Logical_shift*