
Compressed trie dictionaries (english)

Team 10

Xiangjun Meng, Haojie Xu, Boyang Li



Stevens Institute of Technology

May 15th, 2021

"I pledge my honor that I have abided by the Stevens Honor System."

Introduction

Trie is a way of storing and retrieving information.[1] A compressed trie like a tree data structure. In the trie class, adding suffix characters one by one and inserting suffix three letters is two ways we use to set up an English dictionary. For the English dictionary, we also use a hash code to encode the English dictionary characters one by one trie node. The output is a binary file.

Research

Trie Node

The trie data structure has many properties, making it especially attractive for representing large files of data.[2] Fig. 1 shows an example of trie words "at", "all", "cat", "dog". Using Trie,

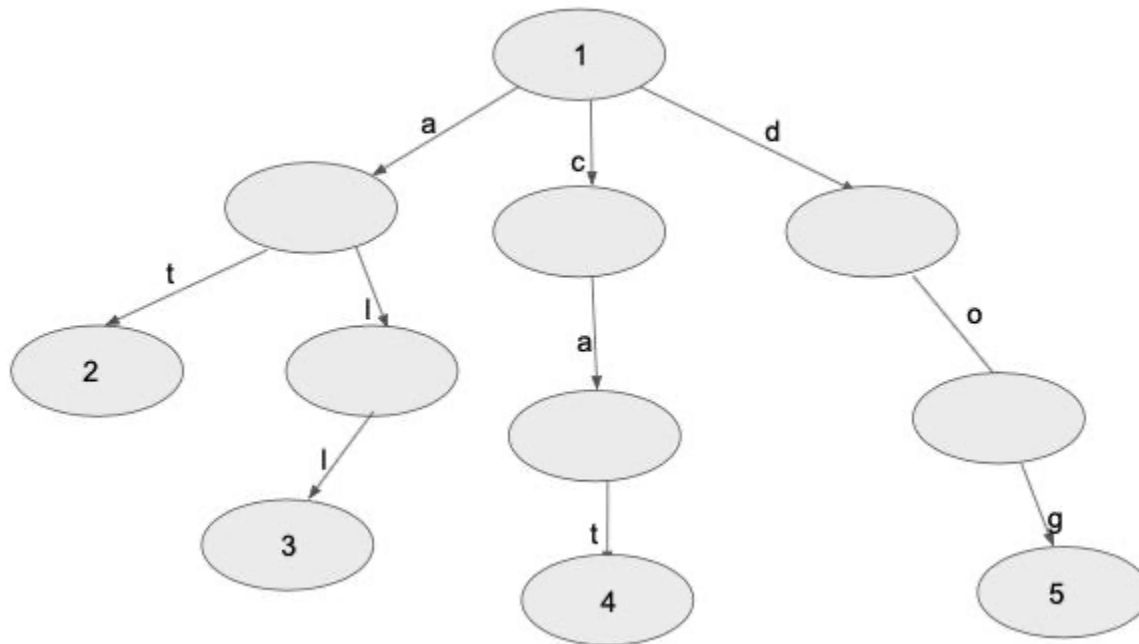


Figure 1: A trie for words "at", "all", "cat", "dog".

the search function in a Java class can be brought to optimal limit (key length). Our project use cases about trie are Strings and Characters typed elements, and fast worst-case search and add functions.

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys.[3] We mark the last node of every key as the end of a word node. A trie node field isWord is worked for distinguishing the node as end of word node. Algorithm 1 represents nodes of the English dictionary characters.

Algorithm 1 Trie Node

```
1: struct TrieNode {  
  
2 :   TrieNode[] children = new TrieNode[26];  
  
3:   char ch;  
4:   boolean isWord;  
5: end
```

Single Character Compress:

The basic process for this is, given a .txt file full of words, the first thing is to load it into the memory, and then the program will add each word to the trie to generate a complete trie. And visit every trienode and convert all trie nodes into bytes and write those bytes into a binary file. The basic operation is as follows:

```
Trie4 trie = new Trie4();  
File file = new File("dict.txt");  
try {  
    Scanner sc = new Scanner(file);
```

```

while (sc.hasNextLine()) {
    String word = sc.nextLine();
    word.trim();
    if(word.length() > 0) {
        trie.add(word);
    }
}
trie.createBinary();
} catch (Exception e){
    System.err.println(e);
}

```

Each line in the file is a word, and then use the Scanner to traverse each line of the file. For each word, use the trim() function to remove the spaces at the beginning and the end, and add this word to the trie.

The trieNode looks like this:

```

class TrieNode{
    TrieNode[] children;
    boolean isword;
    char ch;
}

```

Each trienode contains a char, an array, the size of the array is 26, and one boolean. Each element in the array points to the trienode at the next level, and there is a boolean indicating whether it is a word so far.

After trie is ready, we need to traverse the trie, and then turn each trienode into a byte array. We need 28 bits to represent each trienode. The first bit indicates whether the trienode is a leaf node. The second bit indicates whether

the current trienode is a word or not, and the following 26 bits correspond to the array with length of 26 in the trie node, 1 means there is the next trienode, 0 means there is no next trie node.

A trienode is 28 bits, so it means that a trienode needs 4 bytes (a byte is 8 bits). However, in Java, a byte stores whole numbers from -128 to 127, that is, if one byte is used in binary terms, the range is [1000 0000, 0111 1111]. But because English has 26 letters, each letter has at least one word starting with it, so the binary representation of children[] in the root of trie is 26 bits all 1s. So there will be bytes like “1111 1111” in trienode, which is illegal in java.

The solution to the problem is because each trienode has at least 28 bits, and a byte has 32 bits, so we have 4 free bits to use. For each trienode, we divide 28bits into 4 groups, each group has 7 bits, and then add a bit of 0 in front of each group. Because java can represent from 0000 0000 to 0111 1111, the each generated 4 bytes must be legal. So the following function shows how to convert a single trie node into byte[].

```
private byte[] convert(TrieNode node){
    StringBuilder sb = new StringBuilder();
    boolean isLeaf = true;
    for(int j = 0; j < node.children.length; j++){
        if(node.children[j] != null)
            isLeaf = false;
    }
    if(isLeaf) {
        sb.append('1');
    }
}
```

```

}else {
    sb.append('0');
}

if(node.isword){
    sb.append('1');
}else{
    sb.append('0');
}

for(int j = 0; j < node.children.length; j++) {
    if(node.children[j] == null) {
        sb.append('0');
    }else{
        sb.append('1');
    }
}

// so far, we have 1 + 1 + 26 = 28 bits,
// we need 4 more bits to be 4 bytes

String s = new String(sb);

String[] stringArr = new String[4];

int index = 0;

for(int i = 0; i < 4; i++){
    if(i == 3){
        stringArr[i] = '0' + s.substring(index);
    }else{
        stringArr[i] = '0' + s.substring(index, index+7);
        index += 7;
    }

    byteList.add(stringArr[i]);
}

```

```

byte[] ans = new byte[4];

for(int i = 0; i < 4; i++){
    ans[i] = Byte.parseByte(stringArr[i], 2);
}

return ans;
}

```

The next step is to find a way to visit each trie node in this trie. And after the binary is generated, you can also decode successfully. The method we use is to use Queue in java. Here is the method:

First put root in the queue, and then retrieve how many nodes are in the queue at this time. For loop the size, pop each trie node in the queue, for each popped trienode, first, turn this trienode into a byte [] and write it to the binary file, and then traverse the children array of this node, and put its next trienode into the queue. Repeat this process until the size of the queue is 0.

```

FileOutputStream fos = new FileOutputStream(new File("boyang.dat"));
Queue<TrieNode> q = new LinkedList<>();
q.offer(root);
while(!q.isEmpty()){
    int size = q.size();
    for(int i = 0; i < size; i++){
        TrieNode node = q.poll();
        numOfNodes++;
        byte[] array = convert(node);
        fos.write(array);
        for(int j = 0; j < node.children.length; j++){
            if(node.children[j] != null) {

```

```

        q.offer(node.children[j]);
    }
}
}
}
}

```

The decode method is actually the inverse process of the encode method.

Reading a binary file, every 4 bytes as a group is a trienode. But the first bit of 0 in each byte is meaningless, so we have to discard it, and then concatenate the remaining bits, like this: "010001110101101111101101111". The first 2 bits indicate whether it is a leaf and whether it is a word. In order to convert this binary string to a trienode, we also need to use a queue in java.

For each binary string, look at the 26 bits representing the children[] array. If it is 1, it means that this letter has a trienode. Then create a trienode and put it in the queue. Repeating this procedure until all the bytes are processed.

After compression, the binary file size is 2.9MB, so it is not a very effective compression method.

```

BOYLI-MacBook-Pro:Final boyli$ ls -l boyang.dat
-rw-r--r--  1 boyli  staff  2967344 May 16 11:59 boyang.dat

```

Cause Analysis :

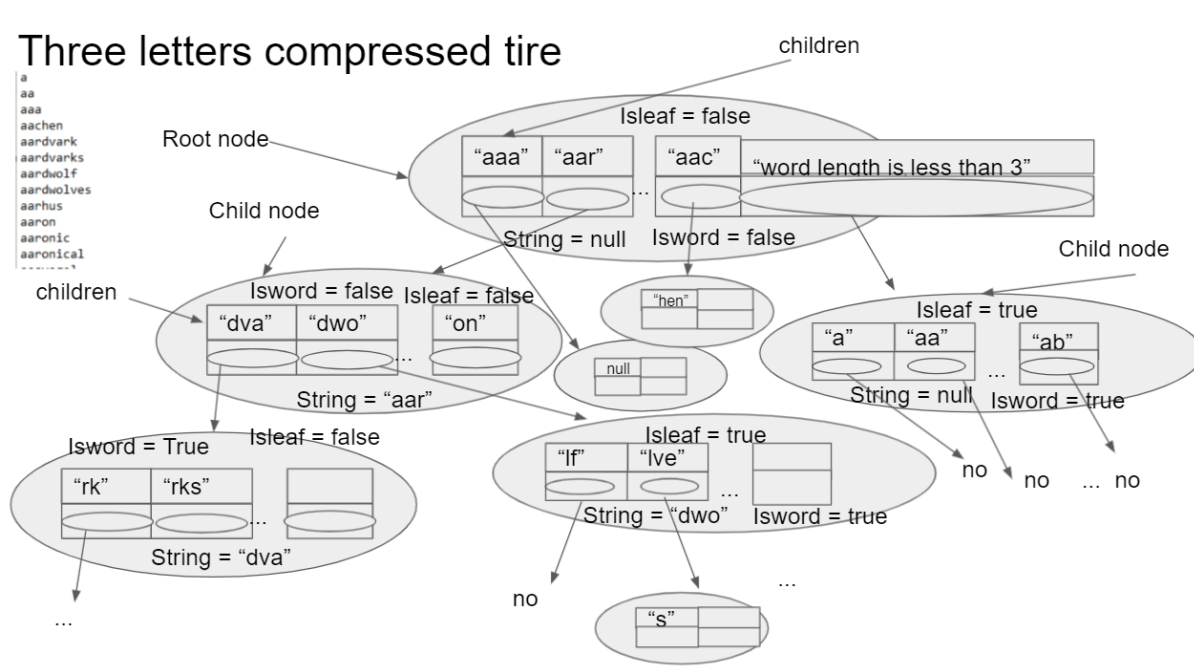
1 For each trienode, 28 bits are minimum required. In order to meet the byte range requirement of the java language, for each node, additional 4 bits of 0 are

added. These 4 bits occupy a lot of extra space, which also reduces the compression efficiency of trie.

2 Then, one single 1 bit can be represented as the leaf node. But if you really use 1 bit to represent the leaf node, there will be a problem. The size of each trie node is different, which leads to that result binary file cannot be decoded. Also, for the operation of binary file writing in the Java language, the smallest unit of operation is byte, so writing single bit is not supported in Java.

Three Character Compress:

This is Three Character Compressed Trie which is different from the single Character Compressed Trie. But, to some degree, they have the same basic rule. As what can be seen from the figure below, it has a clear introduction about what a Three Character Compressed Trie looks like:



Each node of the trie stores 3 letters for one time rather than only one character each time which will save the storage space more. Each node of the trie has exact five objects. First each node has a linkedhashmap which helps to store key of the string word and the value of next level child nodes. Also, each node has 4 flags (one has not been drawn in the figure). String flag is used to depict which this trie node belongs to and the root node of each trie must have a null string flag because the root node will not belong to any string. For example, like what can be seen in the figure, the node in the first position of the second level is from String "aaa" in the root node, so this node just belongs to the string "aaa" and the string flag will be assigned value as "aaa". Isword flag has the function of showing if all the string of all the child node from the root node to current node can compose a whole word like "aap", "end" can compose a whole word. So, the isword flag in the "end" node will have value of true. Apparently, isleaf flag is used to depict whether this node is a leaf node, if it is the flag value will be true otherwise false. The last but not least is the counter which can show the exact order when this word is added into the trie(e.g. the word "append" is the first word added into the trie, it has word_number = 1)and even though when the duplicated word is added into the trie, the counter will not plus 1. So counter can also count how many words has been added into this trie almost like a dictionary.

Meanwhile, Dividing and Storing words when we add or search the exact word will be 3 letters for one time. For instance, if a word which is “append” wants to be added into this trie, first, divide this word in the way of three char by three

```
@Override
public void add(String word){
    int bitNumber = 3; // the number of letter would be stored in one level- one hash
    int remainder = word.length()%bitNumber; // to judge if the word length is the multiple of bitNumber
    int quotient = word.length()/bitNumber; // to judge if the word is less or more than 3
    if(root == null){
        root = new TrieNode2();
    }
}
```

char. So the dividing result is “app”, “end”. For those which word length is less than 3, the root hashmap has a special key “word length is less than 3” and it has a child trie node. In this trie node, there is a special hashmap stores these special strings.

```
if(quotient == 0){ // when quotient is equal to 0, we know that the word length is less than 3, so put it into a special hash
    if(!root.children.containsKey("word length is less than 3")){
        TrieNode2 temp = new TrieNode2();
        temp.children.put(word,new TrieNode2());
        root.children.put("word length is less than 3",temp);
        word_number++;
        temp.isword = true;
        temp.isleaf = true;
    }else{
        if(root.children.get("word length is less than 3").children.containsKey(word)){return;}
        word_number++;
        root.children.get("word length is less than 3").children.put(word,new TrieNode2());
        root.children.get("word length is less than 3").isleaf = true;
        root.children.get("word length is less than 3").isword = true;
    }
}
```

Meanwhile, from the code above, we need to judge which kind the string word belongs to. And then do the following operations. The more detailed code is all in GitHub.

In the second step, String “app” will be added into the root node by using the linkedhashmap function “put(key,value)”, and then the value is just the child trie

```
if(remainder == 0){ // this is the condition when the word length is multiple of 3
    TrieNode2 temp; // we need a pointer to follow the iteration level to keep track of the level where we are now
    temp = root; // each time we add a new word, we will start from the root
}
```

node. Using a pointer temp to follow the exact level where the current level is, so after adding the “app” in the root it can go into the second level. In this child trie node, do the same operation as what did in the root node and the third level and so on so forth till the whole word has been added into the trie at the last step.

```
for(int i = 0; i < quotient; i++){
    String input = (String)word.subSequence(i*bitNumber,(i+1)*bitNumber);
    if(!temp.children.containsKey(input)){ // if the map does not have the string value
        temp.children.put(input,new TrieNode2());
        add_new_word = true;
        temp.children.get(input).ch = input;
    }else{
        while(temp.children.containsKey(input)){ // if it's true,say input exists and the value which is the trienode2 also exists,so t
            temp.isleaf = false;
            temp = temp.children.get(input);
            i++;
            if(i>=quotient){break;}
            input = (String)word.subSequence(i*bitNumber,(i+1)*bitNumber); // if i is larger than quotient, so it will go out of the boundary
        }
        if(i<quotient){
            temp.children.put(input,new TrieNode2());
            temp.children.get(input).ch = input;
            add_new_word = true;
        }
    }
}
```

However, if the word is has been added into the trie in the past, and it wants to be added for one more time, it will start the check operation. For example, like the code above, after dividing the word into several segments, use the hashmap function “containsKey” to judge if this string has already existed in the hashmap, if so, the pointer “temp” will go the next level to check the next 3-letter string and so on so forth. However, if the string is not in the hashmap, so a new key will be added into the hashmap then create a new branch (a new child node). In conclusion, when all word have been added into the whole trie, the trie resembles a tree structure. So the name of the trie is almost from the tree.

As it is like a tree, the adding and searching speed is fast, every time when traversing the trie, the complexity would be $O(\log n)$,at the same time, this kind

of trie also compress the whole dictionary. So from the time complexity and space saving, this kind of structure has both advantages.

```
public String stringSearch(String word){
    int bitNumber = 3; // the number of letter would be stored in one level- one hash
    int remainder = word.length()%bitNumber; // to judge if the word length is the multiple of bitNumber
    int quotient = word.length()/bitNumber; // to judge if the word is less or more than 3
    if(root == null){return "NO";}

    if(quotient == 0){
        if(root.children.containsKey("word length is less than 3")){
            if(root.children.get("word length is less than 3").children.containsKey(word)){return "Yes";}
            return "No";
        }else{
            return "No";
        }
    }else{
        TrieNode2 temp = root;
        int iteration_number = 0;
        for(int i = 0; i < quotient; i++){
            String input = (String)word.subSequence(i*bitNumber, (i+1)*bitNumber);
            if (!temp.children.containsKey(input)){return "No";} // if the hashmap of the node does not contain the 3-letter key, it will return immediately
            temp = temp.children.get(input);
            iteration_number++;
        }if(remainder == 0){
            return "Yes";
        }else{
            String input = (String)word.subSequence(iteration_number*bitNumber, word.length());
            if (temp.children.containsKey(input)){return "Yes";}
            return "No";
        }
    }
}
```

Above is the searching part of the code, the structure of the code is quite like the add part, so no more explanation here. More detail could be seen in code and the commentary in GitHub.

```
public void showTrie(TrieNode2 node) throws IOException {
    Map<String, TrieNode2> map = node.children;
    for (String key: map.keySet()){
        composed_letters.add(key);
        if(node.equals(root)){
            root_letters.add(key);
            System.out.print("\n");
        }
        System.out.print(key+"\t");
        showTrie(map.get(key));
        i++;
    }
}

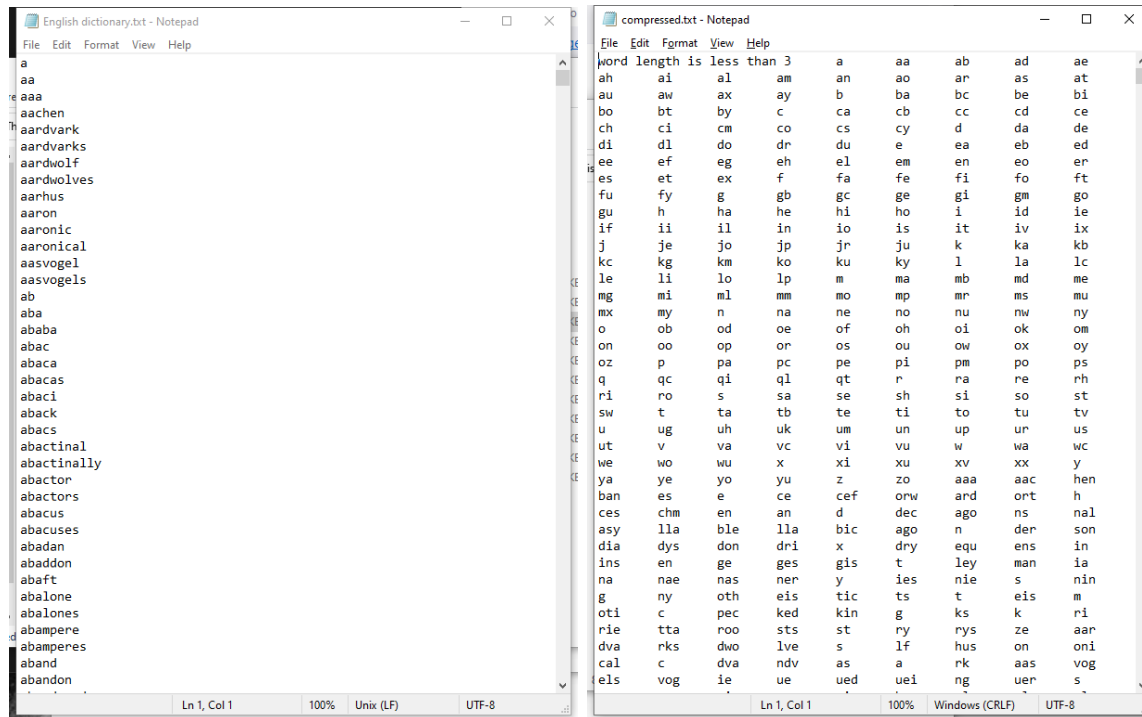
//
```

Trie2

```
word length is less than 3 a aa ab ad ae ah ai al am an ao ar as at au aw ax ay t
aaa
aac |
hen ban es e ce cef orw ard ort h ces chm en an d dec ago ns nal asy lla ble lla bic ago r
aar dva rks dwo lve s lf hus on oni cal c
```

Then the system.out.print shows that the string word are printed branch by branch, which means that it will from the first key of the root node to the

bottom(i.e. the leaf node of this branch) and then return to the root node and next branch, so on so forth. So, it is clear for users to check the word added into the trie.



Above is the result of compressed into txt file, so input the word also branch by branch, which means that after writing into the “word length is less than 3” and the following word, then it will write into the txt file next branch.

English dictionary.txt	5/4/2021 4:56 PM	Text Document	1,864 KB
compressed.txt	5/13/2021 5:40 PM	Text Document	882 KB

From the screenshot above, the first step compressed efficiency is quite good which is more than fifty percent of compressed rate.

97	a	01100001
98	b	
99	c	01100010
100	d	
101	e	01100011
102	f	
103	g	01100100
104	h	
105	i	01100101
106	j	
107	k	01100110
108	l	
109	m	01100111
110	n	
111	o	01101000
112	p	
113	q	01101001
114	r	
115	s	01101010
116	t	
117	u	01101011
118	v	
119	w	01101100
120	x	
121	y	
122	z	

01101101

01101110

01101111

01110000

01110001

01110010

01110011

01110100

01110101 01111000

01111001 01110110

01111010 01110111

Then because a byte word is just one byte, and it can represent an English word regardless of the uppercase or the lowercase and even some special signature only using from -128~127 value range which means that only using the

```
public String strToBinary(String s) // turn the string into the binary string .. need to be further improved to neglect the first "0"
{
    int n = s.length();
    StringBuilder count = new StringBuilder();
    for (int i = 0; i < n; i++)
    {
        // convert each char to
        // ASCII value
        int val = (int) s.charAt(i);

        // Convert ASCII value to binary
        StringBuilder bin = new StringBuilder();
        while (val > 0)
        {
            if (val % 2 == 1)
            {
                bin.append('1');
            }
            else
            {
                bin.append('0');
            }
            val /= 2;
        }
        bin = new StringBuilder(reverse(bin.toString()));

        binary_array.add(Byte.parseByte(bin,2));
        count.append(bin); // use the stringbuilder to form a binary string
        System.out.print(bin + " ");
    }
    binary_array.add(count.toString()); // iteration ends, add the string to the arry
    return count.toString(); // function of returning the binary string according to the ascii code: a--> 01100001
    // binary_array contains the binary string like 01010101
}
```




7 bits (the first digit is always 0 which can be ignored). Basing on this, one method of using arithmetic encoding way can be implemented. First, the string in each node should be transformed into this kind of binary string word. The process can be seen in the screenshot above. The method input is each string and the output is the binary string like from “aaa” to “01100001”, actually there is a mistake here. This function needs to be further improved to return a 7 bits string like “110001” rather than 8 bits string. Or in some other place of the method, this mistake needs to be fixed and the method needs to be further finished.

Because the imminent presentation, basically this need has not been met well.

In the future work, this problem will be solved.

```
public void writeBinaryToTxtfile(TrieNode2 node) throws IOException {
    turnIntoBinary(node);
    FileOutputStream fos = new FileOutputStream(new File( pathname: "compressed1.dat"));
    for(String input:binary_array){
        byte[] data = convert(input);
        for(byte fun:data){
            fos.write(fun);
            System.out.println(fun);
        }
        //        fos.write(data,0, data.length);
    }
    fos.flush();
    fos.close();
}
```

Then the returned binary string is stored in the array of “binary_array”, this binary array will be written into a .dat file and which will do a further compression. Because there is a mistake here needed to be fixed in the future work, the original compression result maybe wrong, but I still put it here just for reference:

 English dictionary.txt	5/4/2021 4:56 PM	Text Document	1,864 KB
 compressed.txt	5/13/2021 5:40 PM	Text Document	882 KB
 compressed1.dat	5/13/2021 5:40 PM	DAT File	621 KB

So from the result, the more compression rate is achieved for about 10% more. This is the three letters compression trie. Meanwhile all the code is written by myself only a little part of citation which has been written in the citation txt file in the GitHub.

Arithmetic Encoding Three Character Compress:

The second encoding method for the trie with 3 letters as the key is to use arithmetic encoding. Arithmetic encoding is a method of turning characters into numbers. For example, the arithmetic encoding of "abc" is $((a-a)*26 + (b-a))*26 + c - a$. Thus, for a 3-letter string, there are $26*26*26$ possibilities. Each trienode is actually a large array of hashset, and the hashset stores all the words with the index of this array as the prefix. To write this array into a binary file, you only need to use `ObjectOutputStream`.

```
FileOutputStream f = new FileOutputStream("boyang_2.dat");
ObjectOutputStream oos = new ObjectOutputStream(f);
oos.writeObject(map);
f.close();
```

Here is the result:

```
BOYLI-MacBook-Pro:Final boyli$ ls -l boyang_2.dat
-rw-r--r--  1 boyli  staff  2469679 May 16 21:05 boyang_2.dat
```

It can be seen that the compression efficiency is not very high. The reason is because, for hashset, if all the words in it become bytes, it is difficult to decode them.

Reference

1. *Trie Memory*, Edward Fredkin, Bolt Beranek and Newman, Inc., Cambridge, Mass.
https://dl.acm.org/doi/pdf/10.1145/367390.367400?casa_token=KO7o2_6anN8AAA:AA:FHT6Sz4zXeyQYeNssX-7NXaQKU6H3ZFLLBkAZbu9ym7qkHDMbnxYN8Dd9nRGDYi1bD9iDZ-a5Sw.
2. *Algorithms for trie compaction*, JUN-ICHI AOE AND KATSUSHI MORIMOTO, Department of Information Science and Intelligent Systems University of Tokushima ,Minami-Josanjima- Cho, T okushima-Shi 770, Japan, AND TAKASHI SATO. Department of Arts and Sciences, Osaka Kyoiku University, 3-1-1, Ikeda 563, Japan. <https://www.co-ding.com/assets/pdf/dat.pdf>
3. <https://www.geeksforgeeks.org/trie-insert-and-search/>